Dependability Assessment of NAND Flash-memory for Mission-critical Applications

Dependability Assessment of NAND Flash-memory for Mission-critical Applications

Michele Fabiano

Michele.Fabiano@polito.it
Michele.Fabiano@gmail.com

Submitted in total fulfillment of the requirements
of the degree of Doctor of Philosophy

February 2013

Faculty of Computer Engineering
Department of Control and Computer Engineering
Politecnico di Torino

Herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This thesis has not previously been presented in identical or similar form to any other Italian or foreign examination board. The thesis work was conducted from 01/2009 to 01/2013 under the supervision of Prof. Paolo Prinetto at Politecnico di Torino.

ABOUT THE AUTHOR

Michele Fabiano received the Bachelor Science (BS) and Master Science (MS) degree in computer science and information engineering from the Universita' degli Studi di Napoli "Federico II", Napoli (Italy), in 2008.

During this PhD activity, he was Visiting Scientist at the European Space Research and Technology Centre (ESTEC), Nordwijk (Netherlands), where he worked on the fault-tolerant oriented design of high performance NAND flash-based architectures in the On Board Computers and Data Handling Section.

He is currently working as Embedded HW/SW Engineer at ASML, Veldhoven (Netherlands).

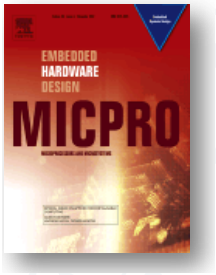
His major research interests are the development of real-time embedded systems, implementation of VLSI for digital signal processing, and development of parallel processing software for automated code generation.

POLITECNICO DI TORINO
1859
1906

LIST OF PUBLICATIONS

ISI journals (accepted for publication)

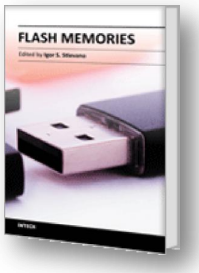
Fabiano M., Furano G., *NAND Flash Storage Technology for Mission-critical Space Applications*, accepted for publication on IEEE Aerospace and Electronic Systems Magazine (AESS).



ISI journals (revisions being processed)

Di Carlo S., **Fabiano M.**, Indaco M., Prinetto P., *Design and Optimization of Adaptable BCH Codecs for NAND Flash Memory*, revisions being processed on Elsevier Microprocessors and Microsystems (MICPRO).

Chapters in edited books



Caramia M., Di Carlo S., **Fabiano M.**, Prinetto P., *Design Issues and Challenges of File Systems for Flash Memories*, Flash Memory, Edited by Igor Stievano, Published by InTech, pp. 28 (pp. 3 – 30) ISBN 978-9-5330-7272-2.

IEEE Conference papers

Zambelli C., Indaco M., Fabiano M., Di Carlo S., Prinetto P., Olivo P., and Bertozzi D., *A Cross-Layer Approach to the Reliability-Performance Trade-Off in MLC NAND Flash Memories*, on Proceedings of Design, Automation and Test in Europe (DATE) 2012, pages 881–886, 12th – 16th March 2012, Dresden (Germany).

IEEE Conference papers (continued)

Di Carlo S., **Fabiano M.**, Piazza R., Prinetto P., *Exploring Modeling and Testing of NAND Flash memories*, Proceedings of 8th IEEE East-West Design & Test Symposium (EWDTS) 2010, 17th – 20th September 2010, St. Petersburg (Russia).

Di Carlo S., **Fabiano M.**, Piazza R., Prinetto P., *EDACs and Test Integration Strategies for NAND Flash memories*, Proceedings of 8th IEEE East-West Design & Test Symposium (EWDTS) 2010, 17th – 20th September 2010, St. Petersburg (Russia).

Caramia M., **Fabiano M.**, Miele A., Piazza R., Prinetto P., *Automated synthesis of EDACs for FLASH Memories with User-Selectable Correction Capability*, Proceedings of IEEE High-Level Design Validation and Test (HLDVT) 2010, 11th – 12th June 2010, Anaheim, California (USA) ISSN: 1552-6674, Print ISBN: 978-1-4244-7805-7.

Caramia M., Di Carlo S., **Fabiano M.**, Prinetto P., *FLARE: a Design Environment for Space Applications*, Proceedings of IEEE High-Level Design Validation and Test (HLDVT) 2009, 4th – 6th November 2009, San Francisco (USA) ISSN: 1552-6674, Print ISBN: 978-1-4244-4823-4.

Caramia M., Di Carlo S., **Fabiano M.**, Prinetto P., *Exploring Design Dimensions in Flash-based Mass-memory Devices*, Proceedings of 4th International Workshop on Software Support for Portable Storage (IWSSPS) 2009, 15th October 2009, Grenoble (France), pp. 43 – 48.

Caramia M., Di Carlo S., **Fabiano M.**, Prinetto P., *Flash-memories in Space Applications: Trends and Challenges*, Proceedings of 7th IEEE East-West Design & Test Symposium (EWDTS) 2009, 18th – 21st September 2009, Moscow (Russia), pp. 429 – 432.

IEEE Conferences Posters

Di Carlo S., **Fabiano M.**, Indaco M., and Prinetto P., *ADAGE: An Automated Synthesis tool for Adaptive BCH-based ECC IP-Cores*, IEEE International Test Conference (ITC) 2012, Anaheim CA, November 4-9, 2012, pp. 15

Conferences without Proceedings

Fabiano M., Furano G. and Magistrati G., *NAND Flash Storage Technology for Mission-critical Space Applications*, in Single Event Effects (SEE) Symposium, April 3-5, 2012, NASA, San Diego

CONTENTS

List of Figures ix

List of Tables xii

1 Introduction 1

1.1 Mission-critical applications . . . 3

1.1.1 An example: the space environment . . . 3

1.2 Thesis organization . . . 5

2 Dependability of NAND Flash Memory: An Overview 9

2.1 Flash memory issues and challenges . . . 10

2.1.1 Technology . . . 10

2.1.2 Architecture . . . 12

2.1.2.1 Examples of NAND Flash Architecture . . . 13

2.1.3 Address translation and boot time . . . 16

2.1.4 Garbage collection . . . 17

2.1.5 Memory wearing . . . 17

2.1.6 Bad block management . . . 18

2.1.7 Error correcting codes . . . 18

2.1.8 Testing . . . 19

2.2 Using flash-memory as Hard Disk (HD) . . . 19

2.3 Flash-memory Reliability Screening . . . 21

2.3.1 Data Retention (*detrapping*) . . . 21

2.3.2 Endurance (*trapping*) . . . 22

3 Modeling and Testing NAND Flash memory 25

3.1 NAND Flash Disturbances . . . 26

3.1.1 Program Disturbances . . . 27

3.1.2 Read Disturbances . . . 30

3.1.3 Over-Erase Disturbance (OED) . . . 32

3.1.4 Over-Program Disturbance (OPD) . . . 32

3.2 NAND Flash Circuit Level Modeling . . . 32

3.2.1 Intra-cell Faults . . . 33

3.2.2 Inter-cells Faults . . . 33
3.3 A Comprehensive Fault Model for NAND flash . . . 34
3.3.1 The BF&D Extended Test Algorithm . . . 36
3.3.2 Algorithm Complexity . . . 37
3.4 To test or not to test: an important remark . . . 37

4 Adaptable Error Correcting Codes Design for NAND Flash memory 39
4.1 Background and related works . . . 41
4.2 Optimized Architectures of Programmable Parallel LFSRs . . . 45
4.3 BCH Code Design Optimization . . . 48
4.3.1 The choice of the set of polynomials . . . 48
4.3.2 Shared Optimized Programmable Parallel LFSRs . . . 52
4.4 Adaptable BCH Encoder . . . 54
4.5 Adaptable BCH Decoder . . . 55
4.5.1 Adaptable Syndrome Machine . . . 56
4.5.2 Adaptable Berlekamp Massey Machine . . . 59
4.5.3 Adaptable Chien Machine . . . 60
4.6 Experimental Results . . . 62
4.6.1 Automatic generation framework . . . 62
4.6.2 Experimental setup . . . 64
4.6.3 Performance evaluations . . . 66
4.6.4 Synthesis Results . . . 68
4.7 A Cross-Layer Approach for New Reliability-Performance Trade-Offs in Multi Level Cell (MLC) NAND Flash Memories . . . 71
4.8 Conclusions . . . 71

5 Software Management of NAND Flash memory: Issues and Challenges 75
5.1 File systems for flash memories . . . 76
5.1.1 Flash file systems in the technical and scientific literature . . . 78
5.1.1.1 eNVy . . . 78
5.1.1.2 Core flash file system (CFFS) . . . 81
5.1.1.3 FlexFS . . . 84
5.1.2 Open source flash file systems . . . 89
5.1.2.1 Yet Another Flash File System (YAFFS) . . . 89
5.1.3 Proprietary FFS . . . 96
5.1.3.1 exFAT (Microsoft) . . . 96
5.1.3.2 XCFiles (Datalight) . . . 96
5.1.3.3 TrueFFS (M-Systems) . . . 96

5.1.3.4 ExtremeFFS (SanDisk) . . . 97
5.1.3.5 OneFS (Isilon) . . . 97
5.1.3.6 emFile (Segger Microcontroller Systems) . . . 97
5.2 Comparisons of the presented FFS . . . 97
5.3 FLARE: a Design Environment for Flash-based Critical Applications . . . 101
5.3.1 FLARE Architecture . . . 101
5.3.1.1 System Configuration Management . . . 102
5.3.1.2 Flash Memory Simulator . . . 103
5.3.1.3 Dependability Evaluation . . . 103
5.3.1.4 Utilities . . . 104
5.3.2 FLARE Technology Roadmap . . . 104
5.3.3 OSs . . . 104
5.3.4 Flash-memory Emulator . . . 105
5.3.4.1 User Level Emulation . . . 105
5.3.4.2 Kernel Level Emulation . . . 106
5.3.5 Workload . . . 106
5.3.6 Interface . . . 107
5.3.7 Core Functions: YAFFS and Partitioning . . . 107
5.3.8 Fault Injector . . . 109
5.3.9 Monitor and Control . . . 109
5.3.10 Snapshots . . . 109
5.4 Wear Leveling Strategies: An Example . . . 110
5.4.1 Circular Buffer Wear Leveling: Modeling and Lifetime Estimation . . . 112
5.4.2 Examples . . . 113

6 A Case Study: the Space Environment 117
6.1 Background . . . 119
6.2 NAND Flash Memory Space-oriented Design . . . 120
6.2.1 Storage Capacity . . . 121
6.2.2 Power Consumption . . . 121
6.2.3 Mass and Volume . . . 121
6.2.4 Performance . . . 121
6.2.5 Lifetime and Reliability . . . 122
6.2.6 Radiation and Error Rates . . . 122
6.2.7 Wrap-up . . . 123
6.3 Sentinel 2 . . . 124
6.3.1 Onboard Data Storage . . . 124
6.3.2 Storage capacity . . . 125

6.3.3 Mass and volume . . . 126
6.3.4 Power consumption . . . 126
6.3.5 Performances . . . 126
6.3.6 Lifetime and reliability . . . 126
6.3.7 Bit Error Rate (BER) . . . 127

A Reliability Overview **131**
A.1 Mean Time Between Failures (MTBF) and Mean Time To Failure (MTTF) . . . 131
A.2 Failure Rate . . . 132
A.3 Failure In Time (FIT) . . . 132
A.4 Reliability Functions . . . 133
A.5 An Example . . . 133

B Flash-memory Dependability: Screening and Qualification **135**
B.1 Screening and qualification parameters . . . 135
B.1.1 Reliability Methodologies . . . 136
B.1.2 Arrhenius plot (*accelerated-temperature data retention*) . . . 137
B.1.3 An example: flash-memory . . . 138
B.2 Failure rate assessment . . . 140

C Principles of Error Correcting Codes **145**
C.1 ECC Principles . . . 145
C.1.1 Error Detection . . . 147
C.1.2 Error Correction . . . 148
C.1.3 Hamming bound . . . 148
C.2 Bose-Chaudhuri-Hocquenhem Codes Design Flow . . . 149
C.2.1 Design Requirements . . . 149
C.2.2 Parameters Evaluation . . . 150
C.2.3 Code Characterization . . . 151
C.2.4 Shortened Codes . . . 152
C.3 Error Detecting and Correcting Codes: The actual trend . . . 152
C.3.1 Examples . . . 153
C.4 Error correcting techniques for future NAND flash memory . . . 155

D List of symbols and acronyms **157**

Bibliography **163**

LIST OF FIGURES

1.1 Sentinel 2 (with courtesy of European Space Agency) . . . 3
1.2 Simulated Sentinel-2 image (with courtesy of European Space Agency) . . . 4

2.1 A possible taxonomy of the management strategies for flash memories . . . 10
2.2 Comparison of SLC and MLC flash memories . . . 11
2.3 SLC NAND Block Architecture . . . 13
2.4 A 256MB Single Plane 2KB-Page SLC NAND Flash Device . . . 14
2.5 A 512MB Dual Plane 2KB-Page SLC NAND Flash Device . . . 14
2.6 A 1GB Dual Plane 2KB-Page MLC NAND Flash Device . . . 15
2.7 A 2GB Dual Plane 4KB-Page MLC NAND Flash Device . . . 15
2.8 High-level Flash-based Hard Disk . . . 16
2.9 (a) High-level and (b) low-level view of flash architecture . . . 20
2.10 Charge Loss Mechanism in NAND Flash . . . 21
2.11 Endurance in NAND Flash . . . 23

3.1 NAND Flash memory Organization . . . 27
3.2 NAND Flash memories Program Disturbances . . . 28
3.3 Program Disturbances in NAND Flash . . . 29
3.4 Read Disturbance in NAND Flash . . . 30
3.5 NAND Flash memory Intra-cell Faults . . . 33
3.6 NAND Flash memory Inter-cells Faults . . . 34
3.7 BF&D Extended Test Algorithm . . . 36

4.1 Architecture of a r-bit PPLFSR with s-bit parallelism. . . . 44
4.2 Example of the resulting PPLFSR (a) and OPPLFSR (b) with 8-bit parallelism for x^{15}, x^{14} and x^{13} of $p_1(x)$ and $p_2(x)$ . . . 47
4.3 High-level architecture of the OPPLFSR . . . 48
4.4 MCI examples of two hypothetical partitions $S_{i,1}$ and $S_{i,2}$ . . . 50
4.5 The MCI Trend of Table 4.2 . . . 52
4.6 The shOPPLFSR architecture is composed by multiple OPPLFSRs . . . 53
4.7 High-level architecture of the adaptable encoder highlighting the three main building blocks and their main connections. . . . 54

4.8 High-level architecture of the adaptable decoder, highlighting the four main building blocks: the adaptable syndrome machine, the adaptable iBM machine, the adaptable Chien machine, and the controller in charge of managing the overall decoding process 56
4.9 Architecture of the adaptable Syndrome Machine 57
4.10 Example of the schema of a byte aligner for $t = 2$ and $s = 8$ 58
4.11 Architecture of the proposed parallel adaptable Chien Machine with parallelism equal to h 61
4.12 BCH codec automatic generation framework. 63
4.13 Percentage of spare area dedicated to parity bits while changing the correction capability of the adaptable codec of Arch. 2 and Arch. 3 68
4.14 Worst case decoding latency for the three architectures considered. 69
4.15 Worst case dynamic power consumption of the three decoders for the three considered architectures. Power is expressed in mW. 70

5.1 Flash Translation Layer and Flash File Systems 77
5.2 Architecture of eNVy 79
5.3 Steps of the eNVy cleaning process 80
5.4 An example of direct (*i-class1*) and indirect (*i-class2*) indexing for a NAND flash 82
5.5 Flexible Cell Programming 84
5.6 The layout of flash blocks in FlexFS 85
5.7 An example of Data Migration 85
5.8 An example of Dynamic Allocation 87
5.9 An example of Wearing Rate Control 89
5.10 The YAFFS Architecture 90
5.11 An Example of YAFFS Operations 91
5.12 An example of Tnode tree for data file 93
5.13 An high-level overview of FLARE Design Environment 101
5.14 A detailed view of FLARE Architecture 104
5.15 A view of the partitioning process (Source Navigator) 108
5.16 A view of the FLARE design environment 110
5.17 A view of the FLARE design environment (2) 111
5.18 A possible graphical report 111
5.19 A 1GB MLC NAND flash device 112
5.20 A 4GB Dual Plane MLC NAND flash device 114

6.1 Sentinel 2 (with courtesy of European Space Agency) 123
6.2 Architecture of the Sentinel 2 MMFU [129] 125

B.1 Arrhenius Diagram of a Floating Gate device 138

B.2 A possible survival function S(t) for flash-memory 139

C.1 General Encoding/Decoding structure of Error Correcting Code 146

C.2 A "0000" codeword after a single-bit error 147

C.3 Generic case Codeword 147

C.4 The wrong "0001" read codeword 148

C.5 BCH Code Design Flow 149

C.6 Examples of Raw BER and Uncorrected BER 150

C.7 ECC Example for point "Large Block..." 153

C.8 Uncorrected BER for different Error Correcting Codes (ECCs) 153

C.9 512B-ECC16 protecting a 2KB page 154

C.10 1KB-ECC16 protecting a 2KB page 154

LIST OF TABLES

1.1 NAND Vs NOR flash-memory . . . 2
1.2 Operations Voltage for Flash Memories . . . 6

2.1 NAND SLC Vs MLC . . . 12

3.1 NAND Flash Memory Disturbances . . . 35
3.2 NAND Flash Memories Circuit Level Faults . . . 35

4.1 An example of the representation of $p_1(x)$ and $p_2(x)$ . . . 46
4.2 An example of Ω_i . . . 51
4.3 Characteristics of the analyzed architectures . . . 66
4.4 Worst case Parity Bits and Encoding/Decoding Latency. sh_{poly} denotes the maximum number of minimal polynomials shared in the shOPPLFSR of the syndrome machine . 67
4.5 Synthesis Results . . . 70
4.6 Minimal polynomials expressed with the corresponding hexadecimal string of coefficients . . . 72
4.7 Generator polynomial expressed with the corresponding hexadecimal string of coefficients . . . 73

5.1 Comparison among the strategies of the presented FFS . . . 98
5.2 Performance comparison among the presented FFS . . . 100

6.1 Comparison of DRAM and NAND flash technology . . . 120
6.2 Sentinel 2 MMFU Requirements . . . 123
6.3 MMFU Storage Features . . . 125

B.1 Charge Loss Mechanisms and Related Activation Energy . . . 137
B.2 Main parameters adopted for flash-memory screening and qualification . . . 140
B.3 Reliability Data 141

C.1 The Hamming distance between pairs of codewords of 4-bit code . . . 146
C.2 BCH code properties . . . 152

Flash memory is a technology that doesn't depend on Moore's Law...this technology should go at least 10 generations!

Gordon Knight, CEO of Nanochip

CHAPTER 1

INTRODUCTION

Contents of this chapter

1.1 Mission-critical applications
1.2 Thesis organization

Flash memory, thanks to the advances in the manufacturing processes, is continuously reducing its typical feature size. E.g., 20nm NAND flash devices are currently available [85]. These advances are producing enormous gains in speed, single chip array sizes, and consequent reduction in power consumption, both in absolute and relative (watt/bit) terms.

The current market provides two major types of flash-memory: NOR and NAND flash-memory. NOR flash-memory is for EEPROM replacement and is more suitable for program execution. NAND flash-memory is more suitable for storage systems [22, 66]. They both exploit the Floating Gate (FG) transistor, but they differ in the way of performing operations and in the interconnections among cells[1]. Table 1.1 briefly sums up the main characteristic of these types of flash-memory.

The main NAND flash merit is the high speed programming/erasing, while the main demerits are the slow random access and the impossibility of byte-programming. At the opposite, the main NOR flash merit is the high speed random access and the possibility

[1] e.g., NAND flash adopts FN-Tunneling effect for program/erase operations and are much denser than NOR flash

	Standby/ Active Power	Cost per bit	R/W/E Speed	Capacity	Erase Cycles	Code Execution	Interface
NAND	Med/Low	Low	Med/High/Med	High	10^5	Hard	I/O-like
NOR	Low/Med	High	High/Low/Low	Low	10^4	Easy	SRAM-like

Table 1.1: NAND Vs NOR flash-memory

of byte-programming. However, NOR flash suffers of slow programming/erasing.

This document addresses only NAND flash-memories. They are increasingly used for data storage both in consumer electronics (e.g., USB flash drives, Solid State Drive (SSD), digital cameras, MP3 players) and mission critical applications, thanks to their:

- **compactness**: NAND flash are much more compact than magnetic HDs;
- **performance**: they are faster than a common magnetic HD (e.g., Read/Write/Erase about $\mu s/\mu s/ms$), providing a higher data throughput;
- **power consumption**: there is no physical movement on the disk (i.e., there are no mechanical parts)[2];
- **shock-resistance**: the resistance to shocks is much higher than magnetic HDs;

NAND flash is the most suitable solution for embedded applications. Although actual magnetic HD are moving toward higher sizes, in order to leverage the yield costs (e.g., 1TB for about 120$), embedded applications usually do not need such a huge memory. E.g., an embedded application like a mobile phone can ask around 16/32GB HD. Applying a magnetic HDs to such a system has two main difficulties: (i) there is no physical space where a magnetic HD can fit; (ii) nowadays it can be really tough or even not possible to find a magnetic HD with that size.

These motivations and many others are pushing for an extensive use of NAND flash memory as mass-memory devices. However, NAND flash research and literature in the mission-critical environment is not as established as in the commercial applications. Mission-critical applications and hi-rel electronics are struggling in keeping the pace with those advances, for multiple reasons.

[2] the lack of mechanical parts, in turn, implies also an higher reliability

1.1 Mission-critical applications

It is a matter of fact that NAND flash memory devices are well established in consumer market. However, it is not true that the same architectures adopted in the consumer market are suitable for mission critical applications like space [16, 18]. In fact, USB flash drives, digital cameras, MP3 players are usually adopted to store "less significant" data which are not changing frequently (e.g., MP3s, pictures, etc.). Therefore, in spite of NAND flash's drawbacks, a modest complexity is usually needed in the logic of commercial flash drives [16]. On the other hand, mission critical applications have different reliability requirements from commercial scenarios. Moreover, they are usually playing in a hostile environment (e.g., the space) which contributes to worsen all the issues [17].

1.1.1 An example: the space environment

Thanks to the experience at the European Space Research and Technology Centre (ESTEC) in Noordwijk, we can provide an example w.r.t. the critical space environment. Fig. 1.1 shows Sentinel-2. It is the first space mission with a flash-based mass-memory device.

Figure 1.1: Sentinel 2 (with courtesy of European Space Agency)

Sentinel-1 is already flying, while Sentinel-2 will fly soon. Once they both are opera-

tional, this pair of satellites will provide global coverage every five days, delivering high-resolution optical imagery for Global Monitoring for Environment and Security (GMES) land and emergency services. Fig 1.2 shows an example of a land monitoring image.

Figure 1.2: Simulated Sentinel-2 image (with courtesy of European Space Agency)

The specific requirements for an avionic application (e.g., Sentinel-2) must determine the final choice of memory used and European Space Agency's duty is to provide new workarounds to known reliability problems to go from (low performance) failure immune systems to (high performance) failure tolerant systems.

Several studies and researches revealed Samsung NAND flash memory to be more suitable than other flash memory to the use in the space environment [62, 100, 101, 102, 119]. However, the choice of a particular technology presents several issues.

The tremendous advances in the manufacturing processes strongly affect the applicability of the studies cited above to the modern mission-critical applications. We need clever solutions to minimize the cost related to upscreening and ruggedisation operations of newer technologies, by exploiting the previous ones.

Space usually incorrectly refers to flash-memory as Non Volatile RAM (NVRAM). Since NAND flashes are not NVRAM, they have different failure modes from Random Access Memory (RAM). Furthermore, space applications have always dealt with RAMs and not often with NAND flash. Therefore, the true risk is that the common fault tolerant techniques (e.g., ECC) of RAMs will be applied to devices (i.e., NAND flash) which have completely different failure modes. As a consequence, proper fault tolerant strategies are

needed.

Each space mission usually provides an ad-hoc solution. Therefore, we need to explore technology-independent techniques to minimize changes with technology and to avoid the rebuilding of the solutions from scratch.

Each mission is usually provided with proprietary solution from external contractors. It is needed a powerful framework to properly validate, verify and, generally speaking, evaluate the proposed solutions. This environment is still missing for NAND flash memory for space applications, because of the marginal and non-critical use of NAND flash in this field. However, this framework is already under development within this PhD activity [17].

1.2 Thesis organization

This thesis presents the results of 3 years of research in the dependability assessment of NAND flash memory for mission critical applications. We aim at providing practical valuable guidelines, comparisons and tradeoffs among the huge number of dimensions of fault tolerant methodologies for NAND flash applied to critical environments. We hope that such guidelines will be useful for our ongoing research and for all the interested readers. The thesis is organized as follows.

Chapter 2
Dependability of NAND Flash Memory: An Overview

Chapter 2 introduces the main issues related to the dependability assessment of NAND flash devices. From a technological standpoint, all NAND flash are not created equal and may differ in cell types, architecture, performance, timing parameters, command set, etc. However, they respect the following general organization. A NAND flash-memory is usually partitioned into blocks. Each block has a fixed number of pages and each page has a fixed size. A block is the smallest unit for erase operations, while read and write operations are done in terms of pages. Therefore, a page can be erased only if its whole corresponding block is erased. In other words, the space already written cannot be overwritten unless it is erased from the flash-memory device. This is one of the main challenging aspects of flash-memories. Moreover, flash-memory wears out after a certain number of erasure cycles. If the erasure cycles of a block exceed this number, it becomes a "bad block" and is not reliable for storing data anymore.

Chapter 3
Modeling and Testing NAND Flash memory

NAND flash operations rely on FN-Tunneling, which is based on probabilistic concepts. It is not possible to determine exactly how many electrons will enter the Floating Gate (FG), but there is a certain probability[3] that a well-defined number of them will do it. FN-Tunneling requires high voltages to work properly (see Table 1.2).

	V_{CG}	V_D	V_S	V_B
Programming	20V	Ground	Ground	Ground
Reading	5V+V_B	~1V	0V	0V
Erasing	-10V+V_B	4-6V	Float	0V

Table 1.2: Operations Voltage for Flash Memories

Such voltages highly stress each cell of the NAND memory. In particular, they affect the quality of the oxide of each cell, which in turn causes disturbances during read, write and erase operations.

Chapter 3 presents a complete overview of disturbances, modeling and testing of NAND flash, in conjunction with a novel comprehensive fault model and the related testing algorithm.

Chapter 4
Error Correcting Codes for NAND flash memory

Fault tolerance mechanisms are systematically applied to increase reliability and endurance of these devices. In particular, since most available parts are COTS, redundancy must be built into the system to ensure its data integrity during operating lifetime. Redundancy, for example, can be built around multi-chip-modules (MCM), which contains duplicates of one die installed in a single package. Furthermore, proper ECCs are needed. Since independent and manufacturer studies showed NAND flash to have random failures, Bose-Chaudhuri-Hocquenhem (BCH), Perfect Difference Cyclic Set, Low Density Parity Check (LDPC) and similar codes may be a suitable choice. However, each ECC is made of several design dimensions. Choosing the most suitable ECC for a specific mission is always a tradeoff among such dimensions.

[3] refer to Schroedinger and Heisenberg equations

Chapter 4 presents an adaptable Bose-Chaudhuri-Hocquenhem (BCH) based design for NAND flash memory. It can dynamically adapt the correcting capability to the specific condition of the memory. The number of parity bits and the decoding complexity are therefore adapted depending on how many errors have to be corrected. An automatic design environment supports the generation of such a architecture. These are two of the most important topics of our ongoing research.

Chapter 5
Software Management of NAND Flash memory: Issues and Challenges

When using a NAND flash as HD, the file management is a challenging issue to address. In fact, flash memories manage data in a completely different manner if compared to magnetic HDs. Operating Systems (OSs) address this issue with two main approaches: (i) block-device emulation, and (ii) Flash File System (FFS). These methodologies are alternatively adopted for compatibility and performance reasons respectively.

Chapter 5 introduces the main software strategies for tackling NAND flash issues. We propose an overall comparison among several FFSs and we present the novel FLash ARchitecture Evaluator (FLARE) design environment, one of the most important topic of our ongoing research.

Chapter 6
A Case Study: the Space Environment

The most critical remark is that, although NAND flash memories provide so many advantages and are a pivotal component in consumer electronics, their use in mission-critical applications (e.g., space) is still under research.

Most of the commercial NAND flash would be deficient in respect of their ability to operate successfully and survive in the space avionics physical environment. For example, commercial systems and components do not often have adequate operating temperature ranges, packaging or radiation harness performance. The availability of fully qualified space memories is not an option for cost, availability, long lead or performance reasons. Commercial Off The Shelves (COTS) NAND flashes need to be adopted. Since most of them are not able to successfully operate and survive in the critical environment, they request additional cost (i.e., upscreening, protection, redundancy, etc.).

However, the most recent studies point out that: (a) TID tend to become less significant because of the very thin high-k oxides; (b) latch-up mechanisms are becoming less severe in terms of survivability of the device but more widespread due to the physical (3D) stacking of the bare chips; thus, in spite of a lower bias voltage, latch-up is still an issue for some devices; (c) in modern NAND flash, SEE are becoming more and more similar to SEFI, thus most SEE can be assimilated to new classes of SEFI errors. We therefore need strategies to tackle them at digital level to increase memory failure tolerance.

All issues are worsened by the aggressive scaling down of NAND flash. They are effectively accelerating Moore's Law, with a scaling factor 2 each 2 years. E.g., 20nm NAND flash devices are currently available [85].

Appendixes A, B and C

Appendix A provides a generic overview of the main reliability concepts. Appendix B presents the most important issues related to the screening and qualification process of NAND flash memory. Appendix C overviews the ECCs design dimensions and issues.

Conclusions

In conclusion, the proposed dependability assessment of NAND flash-based architectures requires both exploring a huge number of design dimensions and evaluating a huge amount of trade-offs among all such dimensions. Moreover, the reliability requirements of critical applications (e.g., space) are much higher than other common applications (e.g., consumer). Therefore, proper methodologies and techniques are needed to accomplish these strict requirements. We hope that this PhD activity will represent a critical contribution to a thorough understanding of the architectural design of NAND flash device within critical environments (e.g., space avionics).

We will try to introduce the concepts required to understand each topic within each chapter. However, topics are usually related among each other. Therefore, some concepts will be re-introduced, when required, in the next chapters.

At least my pencil never crashes!

Unknown

CHAPTER 2

DEPENDABILITY OF NAND FLASH MEMORY: AN OVERVIEW

Contents of this chapter

2.1 Flash-memory issues and challenges
2.2 Using flash-memory as Hard Disk (HD)
2.3Flash-memory Reliability Screening

The increasing demand for high-speed storage capability both in consumer electronics (e.g., USB flash drives, digital cameras, MP3 players, solid state hard-disks, etc.) and mission critical applications, makes NAND flash memories a rugged, compact alternative to traditional mass-storage devices such as magnetic hard-disks.

The NAND flash technology guarantees a non-volatile high-density storage support that is fast, shock-resistant and very power-economic. At higher capacities, however, flash storage can be much more costly than magnetic disks, and some flash products are still in short supply. Furthermore, the continuous downscaling allowed by new technologies introduces serious issues related to yield, reliability, and endurance of these devices [34, 59, 60, 64, 65, 91, 96]. Several design dimensions, including flash memory technology, architecture, file management, dependability enhancement, power consumption, weight and physical size, must be considered to allow a widespread use of flash-based

devices in the realization of high-capacity mass-storage systems [17].

This chapter introduces the main concepts related to the dependability assessment of NAND flash devices. In particular, Section 2.1 presents the most important design dimensions to address when dealing with flash-memory, Section 2.2 explains how to use NAND flash-memory as hard-disks, while Section 2.3 addresses the most recurring concepts and figures of NAND flash data-sheets.

2.1 Flash memory issues and challenges

Although flash memories are a very attractive solution for the development of high-end mass storage devices, the technology employed in their production process introduces several reliability challenges [59, 65, 96]. The so called Flash Translation Layer (FTL) and the native Flash File System (FFS)[1] have to address these problems with proper strategies and methodologies in order to efficiently manage the flash memory device. Fig. 2.1 shows a possible partial taxonomy of such strategies that will be discussed in the sequel of this section [47].

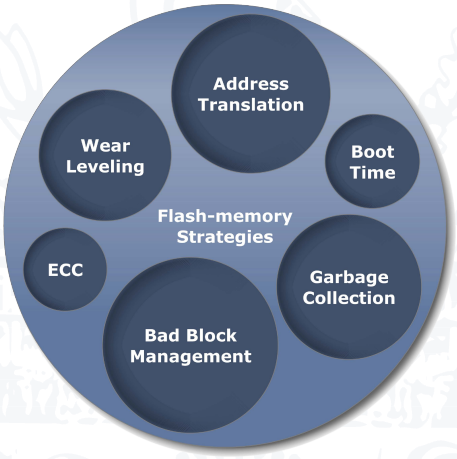


Figure 2.1: A possible taxonomy of the management strategies for flash memories

2.1.1 Technology

The target memory technology is the first parameter to consider when designing a flash-based mass-memory device [18]. The continuous technology downscaling strongly af-

[1] refer to Section 2.2 for more details about FTL and FFS

fects the reliability of the flash memory cells, while the reduction of the distance among cells may lead to several types of cell interferences [64, 91].

From the technology standpoint, two main families of flash memories do exist: (i) NOR flash memories and (ii) NAND flash memories. A deep analysis of the technological aspects of NOR and NAND flash memories is out of the scope of this document[2]. Both technologies use floating-gate transistors to realize non-volatile storing cells. However, the NAND technology allows denser layout and greater storage capacity per unit of area. It is therefore the preferred choice when designing mass-storage systems, and it will be the only technology considered in this document.

NAND flash memories can be further classified based on the number of bit per cell the memory is able to store. Single Level Cell (SLC) memories store a single bit per cell, while Multi Level Cell (MLC) memories allow to store multiple bits per memory cell. Fig. 2.2 shows a comparison between SLC and MLC NAND flash memories [72] considering three main characteristics: capacity, performance and endurance.

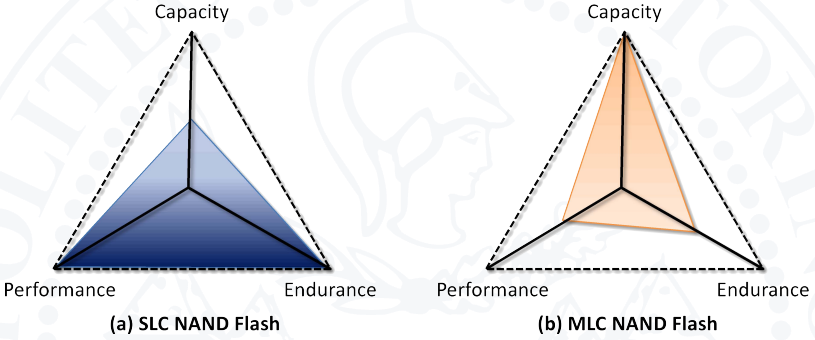


Figure 2.2: Comparison of SLC and MLC flash memories

The MLC technology offers higher capacity compared to the SLC technology at the same cost in terms of area. However, MLC memories are slightly slower than SLC memories. MLC memories are more complex, cells are closer, there are multiple voltage references and highly-dependable analog circuitry is requested [13]. The result is an increased Bit Error Rate (BER) that reduces the overall endurance and reliability [90], thus requiring proper error correction mechanisms at the chip and/or FTL/FFS level.

Consumer electronic products, that continuously demand for increased storage capacity, are nowadays mainly based on MLC NAND flash memories, while mission-critical applications that require high reliability mainly adopt SLC memories [141].

[2] the reader may refer to [60] for additional information

The interested reader may refer to [24, 34] for more detailed comparisons between SLC and MLC technology. Table 2.1 provide some additional figures about these two technologies.

	Features			*Architecture*			*Reliability*			*Array Operations*		
	Bits /cell	**Voltage**	**Bus width**	**Planes**	**Page size**	**Pages /block**	NOP	ECC-**512B**	**Endurance**	t_{READ} **(max)**	t_{PROG} **(avg)**	t_{ERASE} **(avg)**
SLC	1	3.3V, 1.8V	x8, x16	1 or 2	2,112B	64+	1	1	$<10^5$	25us	200 - 300us	1.5 – 2ms
MLC	2+	3.3V	x8	2+	4,314B+	128+	4+	4+	$<10^4$	50us	600 - 900us	2ms

Table 2.1: NAND SLC Vs MLC

Although some entries of the table may not be familiar to the reader, they will be addressed shortly in the sequel of this chapter.

2.1.2 Architecture

The hardware architecture of a NAND flash memory is usually a hierarchical structure organized into cells, pages, blocks and planes.

Memory cells A memory cell is characterized by the so called Floating Gate (FG) inside which electrons can be kept. According to the particular technology chosen, the memory cell can store either 1 bit (i.e., SLC) or 2 bits (i.e., MLC). They will have either 2^1 (i.e., "0", "1") or 2^2 (i.e., "00", "01", "10", "11") voltage reference levels respectively.

Pages A page groups a fixed number of memory cells. It is the smallest storage unit when performing read and programming operations[3]. Each page includes a data area where actual data are stored and a spare area. The spare area is typically used for system level management, although there is no physical difference from the rest of the page. Pages already written with data must be erased prior to write new values. A typical page size can be 2KB plus 64B spare, but the actual trend is to increase the page size up to 4KB+128B and to exploit the MLC technology.

[3]MLC-based devices may allow the so called Partial Page Programming (PPP), which is the ability of programming only a part of the page; a limited Number Of PPP (NOP) per page is allowed; PPP provides more flexibility, but increases the possibility of the so called program disturbances (refer to Chapter 3 for more details about disturbances);

Blocks A block is a set of pages. It is the smallest unit when performing erase operations. Therefore, a page can be erased only if its corresponding block is totally erased. A block typically contains 64 pages, with a trend to increase this number to 128 pages per block, or even more. Since flash memories wear out after a certain number of erasure cycles (endurance), if the erasure cycles of a block exceed this number, the block cannot be considered anymore reliable for storing data. A typical value for the endurance of an SLC flash memory is about 10^6 erasure cycles.

Fig. 2.3 shows the organization of a 128KB+4KB SLC NAND flash-memory block [34]. In this example, we have 2KB+64B per page and 64 pages per block.

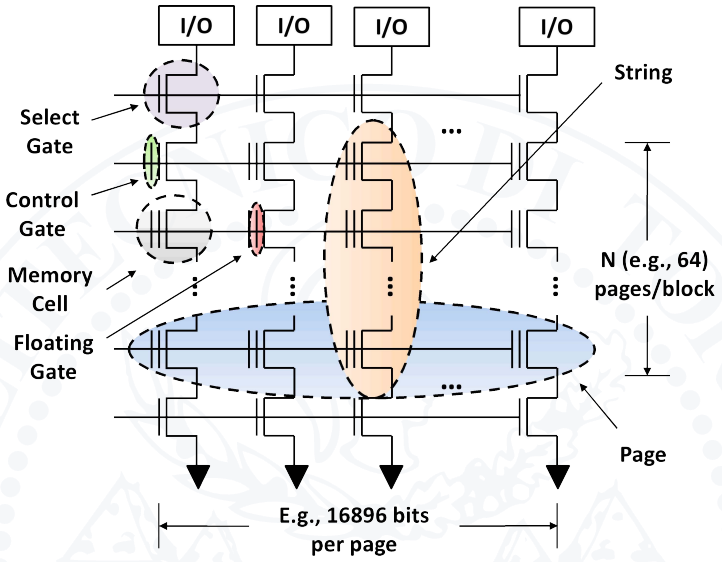


Figure 2.3: SLC NAND Block Architecture

Planes Finally, blocks are grouped into planes. A flash memory with N planes can read-/write and erase N pages/blocks at the same time [34].

2.1.2.1 Examples of NAND Flash Architecture

For sake of completeness, we provide a few examples of NAND flash devices. Examples includes both SLC and MLC technology [34].

256MB Single Plane SLC NAND Device Fig. 2.4 provides the first basic example of a single plane 256MB NAND flash device.

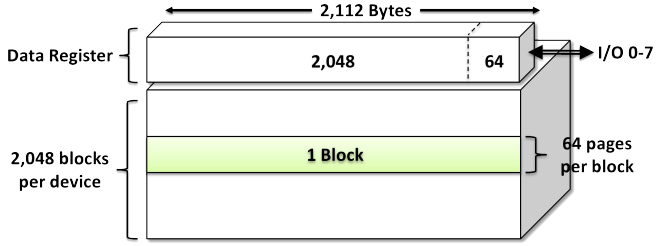


Figure 2.4: A 256MB Single Plane 2KB-Page SLC NAND Flash Device

Each page is 2KB+64B long. SLC technology is used. A data register able to store a full page is provided, and an 8-bit data bus (i.e., I/O 0-7) is used to access stored information.

512MB Dual Plane SLC NAND Device Fig. 2.5 shows an example of a 512MB dual plane SLC NAND flash memory architecture.

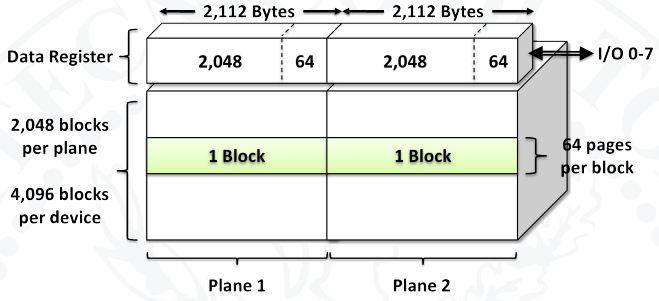


Figure 2.5: A 512MB Dual Plane 2KB-Page SLC NAND Flash Device

The double capacity w.r.t. Fig. 2.4 is accomplished by adding an additional plane[4]. Each plane can store 256MB with pages of 2KB+64B. A data register able to store a full page is provided for each plane, and an 8-bit data bus (i.e., I/O 0-7) is used to access stored information. Doubling the planes basically doubles the operations throughput by allowing:

- Read/Write two pages of different planes at the same time;
- Erase two blocks of different planes at the same time;

1GB Dual Plane MLC NAND Device Fig. 2.6 shows an example of a 1GB dual plane NAND flash device.

[4] Plane 1 and plane 2 are addressing the even- and odd-numbered blocks respectively

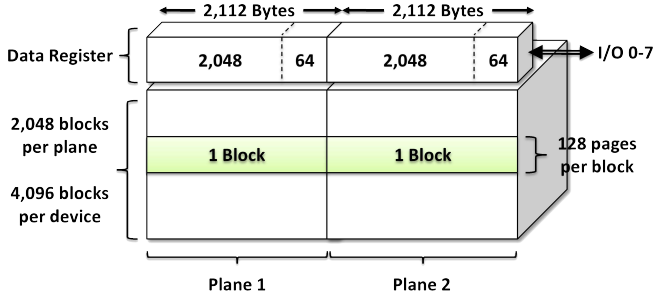


Figure 2.6: A 1GB Dual Plane 2KB-Page MLC NAND Flash Device

The doubled capacity w.r.t. Fig. 2.5 is accomplished by adopting a MLC technology with 2-bit per cell. In order to indicate this change in a simple way, Fig. 2.6 presents a doubled number of pages (i.e., 128). The smart reader would note that the capacity is doubled, while the area is the same of Fig. 2.5.

2GB Dual Plane 4KB-Page MLC NAND Device Fig. 2.7 shows a an example of a 2GB dual plane NAND flash device.

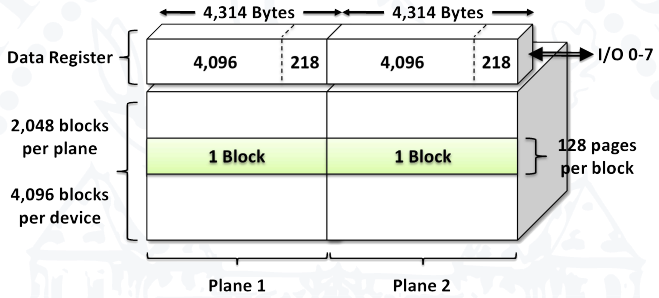


Figure 2.7: A 2GB Dual Plane 4KB-Page MLC NAND Flash Device

The double capacity w.r.t. Fig. 2.6 is accomplished by adopting a page 4KB+218B long. Each plane can store 1GB with pages of 4KB+218B. Data registers are growing accordingly, while an 8-bit data bus (i.e., I/O 0-7) is still used to access stored information.

Note Several variations of this basic architecture can be produced, with main differences in performance, timing and available set of commands [34]. To allow interoperability among different producers, the Open NAND Flash interface (ONFi) Workgroup is trying to provide an open specification (ONFi specification) to be used as a reference for future designs [104]. To March 11, 2013, Samsung and Toshiba are not member of

the ONFi Workgroup. E.g., Samsung is moving toward legacy solutions like the OneNAND™technology [114]. OneNAND will not be addressed in the sequel of this document.

2.1.3 Address translation and boot time

Each page of a flash is identified by both a logical and physical address. Logical addresses are provided to the user to identify a given data with a single address, regardless if the actual information is moved to different physical locations to optimize the use of the device. The *address translation* mechanism that maps logical addresses to the corresponding physical addresses must be efficient to generate a minor impact on the performance of the memory. The address translation information must be stored in the non-volatile memory to guarantee the integrity of the system. However, since frequent updates are performed, a translation lookup table is usually stored in a (battery-backed) RAM (Fig. 2.8), while the flash memory stores the metadata to build this table. The size of the table is a trade-off between the high cost of the RAM and the performance of the storage system.

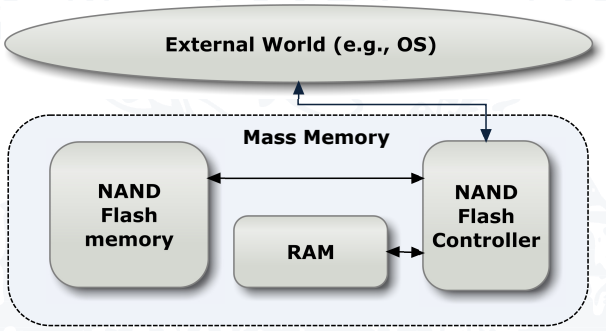


Figure 2.8: High-level Flash-based Hard Disk

Memories with a large page size require less RAM, but they inefficiently handle small writes. In fact, since an entire page must be written into the flash with every flush, larger pages cause more unmodified data to be written for every (small) change. Small page sizes efficiently handles small writes, but the resulting RAM requirements can be unaffordable. At the FTL and at the FFS level, the translation table can be implemented both at the level of pages or blocks thus allowing to trade-off between the table size and the granularity of the table [47].

2.1.4 Garbage collection

Data stored in a page of a flash memory cannot be overwritten unless an erasure of the full block is performed. To overcome this problem, when the content of a page must be updated, the new data are usually saved in a new free page. The new page is marked as *valid* while the old page is marked as *invalid*. The address translation table is then updated to allow the user to access the new data with the same logical address. This process introduces several challenges both at the FTL and at the FFS level.

At a certain point, free space is going to run out. When the amount of free blocks is less than a given threshold, invalidated pages must be erased in order to free some space. The only way to erase a page is to erase the whole block it belongs to. However, a block selected for erasure may contain both valid and invalid pages. As a consequence, the valid pages of the block must be copied into other free pages. The old pages can be then marked as invalid and the selected block can be erased and made available for storage.

This cleaning activity is referred to as *garbage collection*. Garbage collection decreases the flash memory performance and therefore represents a critical aspect of the design of a native flash file system. Moreover, as described in the next subsection, it may impact on the endurance of the device. The key objective of an efficient garbage collection strategy is to reduce garbage collection costs and evenly erase all blocks.

Flexible cleaning algorithms [137], greedy policies, aging functions [28] or periodical collection approaches [130] can be adopted to minimize the cleaning cost.

2.1.5 Memory wearing

As previously introduced, flash memories wear out after a certain number of erasure cycles (usually between 10^4 and 10^5 cycles). If the number of erasures of a block exceeds this number, the block is marked as a *bad block* since it cannot be considered anymore reliable for storing data. The overall life time of a flash memory therefore depends on the number of performed erasure cycles. *Wear leveling* techniques [21, 23, 28, 35, 112] are used to distribute data evenly across each block of the entire flash memory, trying to level and to minimize the number of erasure cycles of each block. The alternative is to consider higher capacity flash-memory devices, taking care of the resulting drawbacks in terms of weight and volume [20].

There are two main wear leveling strategies: dynamic and static wear leveling. The *dynamic* wear leveling only works on those data blocks that are going to be written, while

the *static* wear leveling works on all data blocks, including those that are not involved in a write operation. Active data blocks are in general wear-leveled dynamically, while static blocks (i.e., blocks where data are written and remain unchanged for long periods of time) are wear-leveled statically. The dynamic and static blocks are usually referred as *hot* and *cold* data, respectively. In MLC memories it is important to move cold data to optimize the wear leveling. If cold data are not moved then the related pages are seldom written and the wear is heavily skewed to other pages. Moreover, every read to a page has the potential to disturb data on other pages in the same block. Thus continuous read-only access to an area can cause corruption, and cold data should be periodically rewritten [77].

Wear leveling techniques must be strongly coupled with garbage collection algorithms at the FTL and at the FFS level. In fact, the two tasks have in general conflicting objectives and the good trade-off must be found to guarantee both performance and endurance. The interested reader may refer to [21] for a comparative analysis of the most used wear leveling algorithms.

2.1.6 Bad block management

As discussed in the previous sections, when a block exceeds the maximum number of erasure cycles, it is marked as a *bad block*. Bad blocks can be detected also in new devices as a result of blocks identified as faulty during the end of production test.

Bad blocks must be detected and excluded from the active memory space. In general, simple techniques to handle bad blocks are commonly implemented. An example is provided by the Samsung's XSR (Flash Driver) and its Bad Block Management scheme [111]. The flash memory is initially split into a reserved and a user area. The reserved blocks in the reserved area represent a *Reserve Block Pool* that can be used to replace bad blocks. Samsung's XSR basically remaps a bad block to one of the reserved blocks so that the data contained in a bad block is not lost and the bad block is not longer used.

2.1.7 Error correcting codes

Fault tolerance mechanisms and in particular Error Correcting Code (ECC) are systematically applied to NAND flash devices to improve their level of reliability. ECCs are cost-efficient and allow detecting or even correcting a certain number of errors.

ECCs have to be fast and efficient at the same time. Several ECC schema have been proposed based on linear codes like Hamming codes [84] or Reed-Solomon (RS) codes [115, 127]. Among the possible solutions, Bose-Chaudhuri-Hocquenhem (BCH) codes are linear codes widely adopted with flash memories [33, 45, 67, 81]. They are less complex than other ECCs, providing also a higher code efficiency. Moreover, manufacturers' and independent studies [42, 48, 138] have shown that flash memories tend to manifest non-correlated bit errors. BCH are particularly efficient when errors are randomly distributed, thus representing a suitable solution for flash memories.

The choice of the characteristics of the ECC is a trade-off between reliability requirements and code complexity, and strongly depends on the target application (e.g. consumer electronics vs mission-critical applications) [18].

ECC can be implemented both at the software-level, or resorting to hardware facilities. Software implemented ECC allow to decouple the error correction mechanisms from the specific hardware device. However, the price to pay for a software-based ECC solution is a drastic performance reduction. For this reason, available file systems tend to delegate the code computation tasks to a dedicate hardware limiting the amount of operations performed in software, at the cost of additional resources (e.g., hardware, power consumption, etc.) and reduced flexibility.

The interested reader may refer to Appendix C for more details about ECCs and BCH. Furthermore, Chapter 4 is addressing the design and the practical implementation of an adaptable BCHs for NAND flash-memory.

2.1.8 Testing

Flash-memory testing is quite different from testing other kinds of memory. In fact, flash-memory experiences disturbances or faults not conforming to any of the traditionally known fault models used in testing RAMs. Firstly, we need specific fault models to properly represent the most frequent physical defects. Then, we are able to devise efficient test algorithms to test them [18].

Chapter 3 about testing flash-memory is thoroughly addressing such a peculiar issue.

2.2 Using flash-memory as Hard Disk (HD)

Among the different issues to consider when designing a flash-based mass-storage system, the file management represents a challenging problem to address [47]. In fact, flash

memories store and access data in a completely different manner if compared to magnetic disks. This must be considered at the OS level to grant existing applications an efficient access to the stored information. Two main approaches are pursuit by OSs and flash memory designers: (i) block-device emulation, and (ii) development of native file systems optimized to operate with flash-based devices [22, 47]. Both approaches try to address the issues discussed in Section 2.1.

Block-device emulation refers to the development of a hardware/software layer able to emulate the behavior of a traditional block device such as a hard-disk, allowing the OS to communicate with the flash using the same primitives exploited to communicate with magnetic-disks. This layer is usually referred as Flash Translation Layer (FTL). Fig. 2.9 (left) shows a very high-level view of a FTL. The FTL "translates" the typical system calls (e.g., open, read, write) of the OS into the proper sequence of commands for the specific flash-memory chip. Fig. 2.9 (right) shows an example of the low-level commands a flash-memory chip [113].

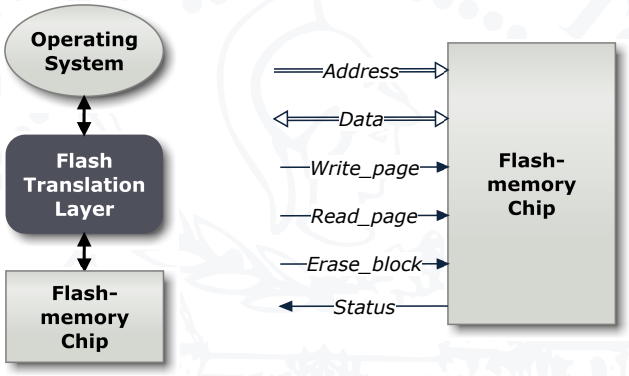


Figure 2.9: (a) High-level and (b) low-level view of flash architecture

The main advantage of this approach is the possibility of reusing available file systems (e.g., FAT, NTFS, ext2) to access the information stored in the flash, allowing maximum compatibility with minimum intervention on the OS. However, traditional file systems do not take into account the specific peculiarities of the flash memories, and the emulation layer alone may be not enough to guarantee maximum performance.

The alternative to the block-device emulation is to exploit the hardware features of the flash device in the development of a *native Flash File System (*FFS*)*. An end-to-end flash-friendly solution can be more efficient than stacking a file system designed for the characteristics of magnetic hard-disks on top of a device driver designed to emulate

disks using flash memories [52]. For efficiency reasons, this approach is becoming the preferred solution whenever embedded NAND flash memories are massively exploited [7, 132, 133].

The literature is rich of strategies involving block-device emulation [22, 23, 61, 66, 75]. [47] offers a comprehensive comparison of available native FFS. Furthermore, Chapter 5 discusses how to properly address the issues of using NAND flash memories as mass-memory devices from the software standpoint.

2.3 Flash-memory Reliability Screening

We think it is worthy to briefly introduce a pair of concepts that will often recur during our discussions: the *data retention* and the *endurance* of a NAND flash device.

2.3.1 Data Retention (*detrapping*)

A flash-memory basically works on a Floating Gate (FG) transistor [59]. The programming operation inject electrons in the FG, while the erase operation does the opposite operation. However, as we can imagine, the FG is subject to wearing and damages [34]. Therefore, as Fig. 2.10 shows, charge loss can occur.

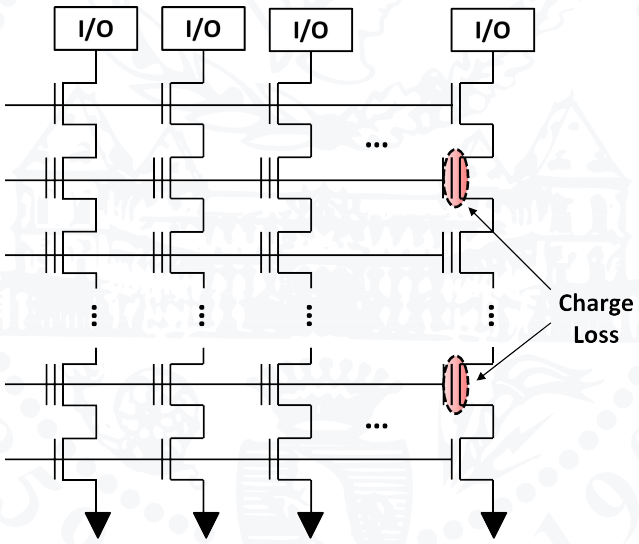


Figure 2.10: Charge Loss Mechanism in NAND Flash

This phenomenon is usually referred as *de-trapping*. It basically causes a shift of the threshold voltage V_{th}, which in turn effectively disturbs the value measured by the sense

amplifiers during read operations.

The *data retention* refers to the ability to maintain stored data between the time of writing and subsequent reading of the stored information. For NAND flash, the data retention time is usually referred as detrapping time t_{det} (i.e., the time needed by "enough" electrons to exit the FG).

Companies usually state in the data-sheets of their flash-memory that, e.g.,

> "*(...) Reliable CMOS Floating-Gate Technology (...) Data Retention: 10 Years...*" [113].

Since, for obvious reasons of time-to-market, it is not feasible to test the flash-memory for 10 years, accelerated strategies are performed [13]. The interested reader may refer to Appendix B for more information about the screening process of NAND flash-memory.

It is important to point out that the memory cell is not damaged. The block, to which the cell/page belongs to, can be cycled (i.e., erased and re-programmed). In order to improve data retention, it is advisable:

- limiting Program/Erase (P/E) Cycles of a block requiring a high retention;
- limiting read operations as much as possible to reduce the Read Disturbance (RD)[5];

Therefore, leveling the operations is the most suitable solution for improving the data retention. This is why wear leveling techniques play a fundamental role for accomplishing a high level data retention for our NAND flash device.

2.3.2 Endurance (*trapping*)

Cycling (i.e., continuously performing Program/Erase operations) has the inconvenient side effect of *trapping* electrons in the dielectric [34, 59]. This phenomenon, as Fig. 2.11 shows, causes an irreversible shift of the threshold voltage V_{th}.

From a circuitry logic standpoint, it can be observable as a failed program or erase operation. The cell is actually physically damaged and cannot be repaired. Therefore, we have to retire the block to which the cell/page belongs to, by marking it as a bad block.

In their data-sheets, companies usually refer to the *endurance* as the number of P/E cycles after which a block of their flash-memory cannot store data in a reliable mode anymore. E.g., Samsung states that their (old) K9XXG08UXM has:

[5] refer to Chapter cha:Testing-NAND-Flash-memory for more details about RD

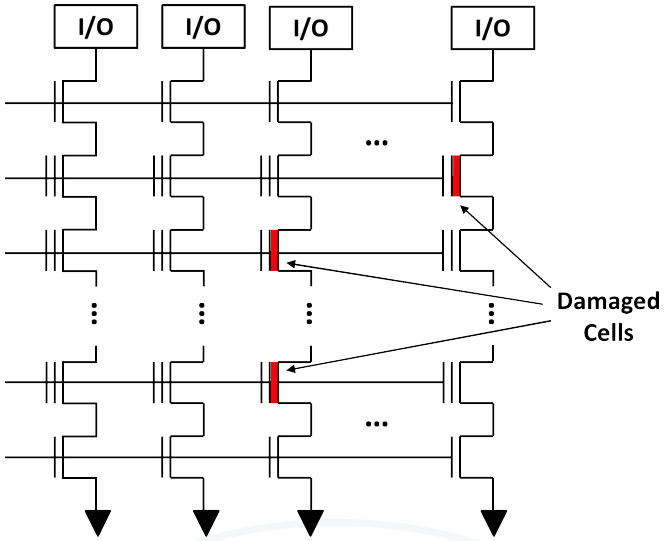


Figure 2.11: Endurance in NAND Flash

"(...) Endurance: 5K Program/Erase Cycles (with 4bit/512byte ECC)..." [113]

In order to improve the endurance of each block, it is advisable:

- checking that program/erase operations did not fail (i.e., $SR0 = pass/fail$ flag set by P/E operations);
- if the program set $SR0 = fail$, moving valid data of current block A to another block and, then, marking the old block as *bad*;
- leveling the wearing of each block of the memory (i.e., Wear Leveling);
- using ECCs both for protecting data and for metadata (i.e., ECCs on the spare area);

Therefore, wear leveling techniques and ECCs to user/spare data are the most suitable methodologies for improving NAND flash endurance.

SUMMARY

This chapter introduced the main concepts related to the dependability assessment of NAND flash devices.

NAND flash-memory experiences phenomena not conforming to any of the traditionally known technology. Therefore, we addressed the peculiar issues and challenges of NAND flash.

Using NAND flash-memory as an hard-disk implies to develop either a Flash Translation Layer (FTL) or a Flash File System (FFS). Both methodologies have to implement specific strategies (e.g., wear leveling, ECC) to address the issues discussed above. The complexity of the applied strategies is strictly related with the complexity of the target application. E.g., although they are based on the same technology, a MP3 player will not require the same dependability of a Solid State Drive (SSD) for space applications.

Data retention and endurance are pivotal recurring concepts in NAND flash (data-sheets). Wear leveling techniques and ECCs play a fundamental role for accomplishing high levels of data retention and endurance.

The problem with troubleshooting
is that trouble shoots back.

Unknown

CHAPTER 3

MODELING AND TESTING NAND FLASH MEMORY

Contents of this chapter

3.1NAND Flash Disturbances
3.2NAND Flash Circuit Level Modeling
3.3A Comprehensive Fault Model and Test Algorithm for NAND flash
3.4To test or not to test: an important remark

Flash-memory testing is quite different from testing other kinds of memory. During read/write/erase operations, flash memories can experience disturbances or faults that do not conform to any of the traditionally known fault models used in testing RAMs [18]. The rationale of flash-memory lies in the Floating Gate (FG) transistor[1]. It is basically a completely insulated metal layer in which electrons can be trapped thanks to the *tunneling* effect. However, this layer is intrinsically not perfectly reliable and leads to problems in terms of *data retention* and *endurance*[2]. Therefore, the reliability issues of NAND flash memory are due to two main reasons: (i) the technology of the FG; (ii) the strong scaling down of technology. The high dependence on technology implies an high complexity of the testing process. Furthermore, the cost-per-cell pushes for a continuous scaling down of technology, which has the side effect of reduc-

[1] a detailed dissertation about FG-based devices is out of the scope of this chapter. The interested reader may refer to [59, 60, 64, 143] for more detailed information about this topic

[2] refer to Section 2.3 for more details

ing the overall endurance and reliability [18, 90]. A complete technology-independent test is therefore needed.

Literature is rich of approaches for NOR flash-memory. A first possible approach is to adopt a functional/logical model of NOR flashes. [27, 30, 65, 93, 94, 96]. However, the alternative is to model flashes with a defect-based approach [70]. However, both approaches usually propose test algorithm(s) related to their models.

Literature about NAND is not as rich as NOR flash. Although these approaches can be extended to NAND flash memories, other methodologies were addressed. [95] provides a simplified list of fault models for NAND flash memories, according to technological assumptions, and the related test algorithm. [58] presents a possible Bridging Fault (BF) model for NAND flash memories.

All these approaches lack considering all the realistic possible permanent faults. Section 3.1 is a complete analysis of the possible disturbances of NAND flash-memory. The principles of the main operations are discussed too. Section 3.2 addresses the modeling of NAND flash from a circuit level standpoint. Section 3.3 proposes a comprehensive fault model and the related test algorithm which is able to test all the possible fault models of NAND flash independently on the specific technology. Finally, Section 3.4 presents some useful remarks about testing NAND flash devices.

3.1 NAND Flash Disturbances

Disturbances are faulty behaviors resulting from the FG technology [59]. As a consequence, they do belong to flash memories, but not to the other memories. The most significant ones are derived from [59] and include: (i) program disturbance faults; (ii) Read Disturbance (RD) faults; (iii) Over-Erase Disturbance (OED) and Over-Program Disturbance (OPD) faults[3].

All these disturbances are able to modify the original value stored inside a cell into another one.

Reference NAND Organization In the sequel, we will refer to the basic organization of Fig. 3.1 to explain the main operations of NAND flash-memory and the related issues.

[3] Erase disturbance faults (i.e., disturbance faults on erase operations) are considered within NOR testing as Word-line Erase Disturbance (WED) and Bit-line Erase Disturbance (BED) [92], but they are usually excluded from NAND testing [97]; they will not be considered in the sequel of this chapter

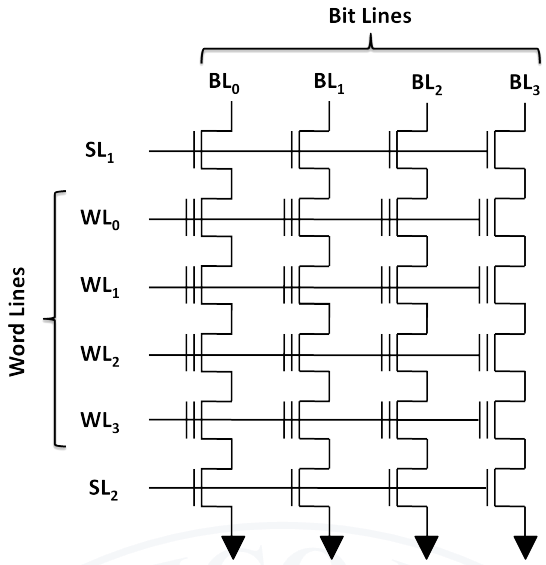


Figure 3.1: NAND Flash memory Organization

Word-Line (WL) is the horizontal line. Bit-Line (BL) is the vertical line. Control gates are connected to WL. BL connects drains together and basically represent data bus. Select Line$_1$ (SL$_1$) and Select Line$_2$ (SL) connects drains and sources to power supply and to common ground respectively. The voltage combination applied to WL and BL define an operation (i.e., read, erase or program).

During program and erase operations, high voltages are applied to the WL and the BL. Therefore, disturbances along them are more critical in flash memory w.r.t. RAMs [139].

3.1.1 Program Disturbances

The state of an erased cell is logically "1". Programming a single NAND flash-memory cell consists into logically writing a "0". Erasing a cell, means logically writing a "1" again.

Unlike NOR and RAMs are *random* access memory, NAND flash memory are referred as *sequential* access memory. This means that, in order to access (to read and write) a cell, we need to "pass" through the others, stressing them.

Fig. 3.2 shows the programming a single cell C11.

When a single NAND flash memory cell is being programmed (i.e., $1 \rightarrow 0$ transition), all the cells in the row (i.e., Word-Line or WL) are subject to a high control gate voltage and all the cells in the same column (i.e., Bit-Line or BL) are biased to be in the pass-

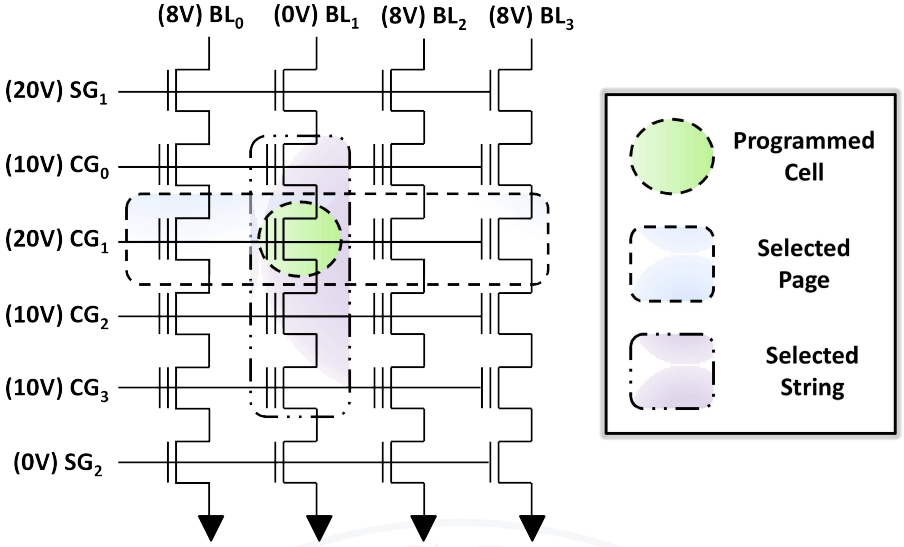


Figure 3.2: NAND Flash memories Program Disturbances

transistor state. This situation can produce unintentional transitions in any of the cells in the WL and/or in the BL of the one being programmed.

WL_1 is subject to high-voltage (e.g., 20V). All the cells of BL_1 become pass-transistors. In this case, program disturbance faults can occur in: (i) a cell sharing the common WL_1; (ii) a cell sharing the common BL_1. Fig. 3.2 refers to them also as *Selected Page* and *Selected String* respectively. Program disturbances can occur only within the block to which the page under program belongs to [34].

According to its initial content, the faulty cell can be programmed or erased. Literature commonly refers to the transition $1 \rightarrow 0$ (i.e., unintentional programming) as:

- Word-line Program Disturbance (WPD) [30, 65, 92, 93, 139] or DC-Programming ($DC-P$) [96]: the selected cell under program causes an unselected unprogrammed cell on the same WL to be programmed; in Fig. 3.2, each unprogrammed cell of the *Selected Page* can be unintentionally programmed;
- Bit-line Program Disturbance (BPD) [30, 65, 92, 93, 139]: the selected cell under program causes an unselected unprogrammed cell on the same BL to be programmed; in Fig. 3.2, each unprogrammed cell of the *Selected String* can be unintentionally programmed;

Literature commonly refers to the transition $0 \rightarrow 1$ (i.e., unintentional erasure) as:

- WED [30, 65, 92, 93, 139] or DC-Erase (DC – E) [96]: the selected cell under program causes an unselected programmed cell on the same WL to be erased; in Fig. 3.2, each unprogrammed cell of the *Selected Page* can be unintentionally erased;

- BED[30, 65, 92, 93, 139] or Drain Disturbance (DD) [96]: the selected cell under program causes an unselected unprogrammed cell on the same BL to be erased; in Fig. 3.2, each unprogrammed cell of the *Selected String* can be unintentionally erased;

Fig. 3.3 provides a more generic example of program disturbances for NAND flash.

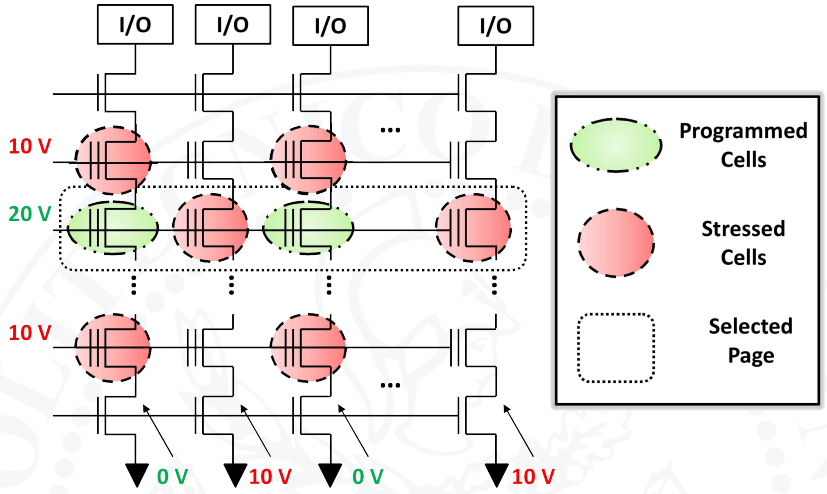


Figure 3.3: Program Disturbances in NAND Flash

Fig. 3.3 shows the programming of two cells (i.e., *Programmed Cells*) which may disturbance the other cells on the same WL or BL of the programmed ones (i.e., *Stressed Cells*)

However, as the name suggests, this phenomenon is only a disturbance. As such, it does not damage cells but simply interferes with their content [34].

Finally, some NAND devices are allowing the so called Partial Page Programming (PPP). It is the ability of programming only part of a page. This ability enables higher flexibility, but increases the chance of program disturbances.

Reducing Program Disturbance In order to leverage the program disturbance phenomenon, it is advisable to:

- program in a sequential way the pages belonging to the same block (e.g., from 0 to 63 for SLC, from 0 to 127 of MLC);
- limit PPP as much as possible;
- program in "one-shot" MLC-based pages;
- adopt ECC strategies to recover from disturbances; e.g., 512B-ECC1[4] per SLC page or at least 512B-ECC16 per MLC page;

3.1.2 Read Disturbances

Being a sequential access memory, NAND flash are stressing (many) unselected pages for reading just one page. Fig. 3.4 shows the read operation of a NAND flash page.

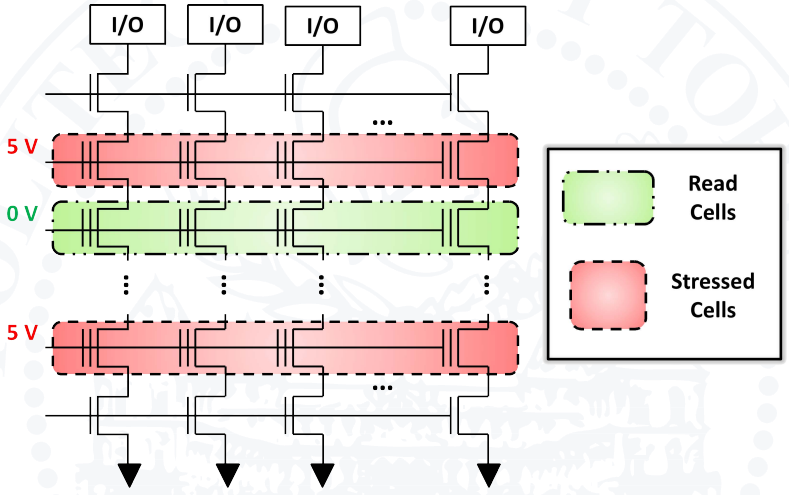


Figure 3.4: Read Disturbance in NAND Flash

The selected page is biased with a defined control gate voltage (e.g., 0V), whereas all the other unselected pages are turned into pass-transistors with a higher control gate voltage (e.g., 5V). A sufficient number of read operations[5] performed on the same page is able to produce unintentional transitions in the page being read. Furthermore, also the other unselected pages may be disturbed.

[4] it means 1-bit correctable (i.e., 1 error tolerated) each 512Bytes
[5] this figure is strictly linked with technology

If after consecutive reads the selected page may change its state, then a Read Disturbance (RD) occurred. Read disturbances can occur: (i) only within the block to which the page being read belongs to; (ii) only in the unselected pages [34].

The RDs for NAND flash memories are well known in literature [65, 97, 139] as:

- RDA(E): the selected programmed cell is read and its content is erased;
- RDA(P): the selected erased cell is read and is programmed;
- RDU(E): the selected cell is read and another unselected programmed cell is erased;
- RDU(P): the selected cell is read and another unselected erased cell is programmed;

However, as the name suggests, this phenomenon is only a disturbance. As such, it does not damage cells but simply interferes with their content [34].

Reducing Read Disturbance To leverage the read disturbance phenomenon, it is advisable to [34]:

- (if possible) level read operations on pages;
- adopt a RDs counter for each block;
- set an ECC emergency threshold;
- as a "rule of thumb", limit to 10^6 (SLC) and 10^5(MLC) the reads of each block;
- erasing a block "reset" its RD count; when either the ECC threshold or the "rule of thumb" is exceeded, we move valid data to a free block and we erase the old one;
- adopt ECC strategies to recover from disturbances; e.g., 512B-ECC1[6] per SLC page or at least 512B-ECC16 per MLC page;

[6] it means 1-bit correctable (i.e., 1 error tolerated) each 512Bytes

3.1.3 Over-Erase Disturbance (OED)

When a flash memory block is erased, all the electrons trapped in the FGs of the cells of a block are simultaneously removed. However, it is important to remark that:

"...not all the cells have equal yield or identical physical conditions..."[34]

Therefore, there may be some cells that are already erased before (i.e., "faster" than) the others. These cells will have a net positive charge in the FG, resulting in a very low threshold [59]. This phenomenon is referred as Over-Erase Disturbance (OED). It is difficult to program cells affected by OED, because they need more program cycles than usual. The result is that automatic *Program&Verify* operations are slowed-down.

3.1.4 Over-Program Disturbance (OPD)

At the opposite, when a page is programmed, there may be some cells that are already programmed before (i.e., "faster" than) the others. These cells will have excessive negative charge in the FG, resulting in a very high threshold [59]. This phenomenon is referred as Over-Program Disturbance (OPD). It is difficult to erase cells affected by OPD, because they would need more erase cycles than usual.

Let us point out another important thing. A cell affected by OPD can prevent the correct reading of other cells. E.g., let consider Fig. 3.4 and assume that there is an over-programmed cell on a particular BL. We want to read another cell in the same BL of the over-programmed cell. Note that all the cells on the same BL are connected in series and the over-programmed cell is behaving as an open circuit (i.e., absence of current detected). Therefore, the result of each read operation on a cell on the same BL will produce always a logic zero.

3.2 NAND Flash Circuit Level Modeling

An alternative approach is to model physical defects of flash memory at the circuit level (i.e., as resistors and capacitors). There are three main contributions to all the possible defects of NAND flash memory: (i) intra-cell faults, (ii) inter-cells faults and (iii) cell to cell interferences [58, 64, 91].

3.2.1 Intra-cell Faults

Fig. 3.5 shows the shorts within a Floating Gate transistor cell, i.e., Control Gate (CG), Floating Gate (FG) , Drain (D), Source (S) and Bulk (B). In particular, the possible shorts are between CG-FG, FG-D, FG-S, FG-B, CG-D, CG-S and D-S.

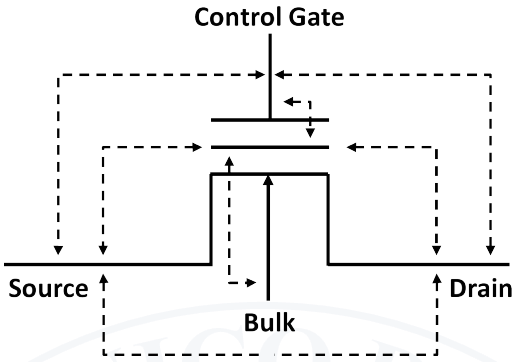


Figure 3.5: NAND Flash memory Intra-cell Faults

It can be shown that all the bridging faults of Fig. 3.5 are equivalent to a Stuck-At Fault behavior. Therefore, they are reported in Table 3.2 as SAF.

3.2.2 Inter-cells Faults

Fig. 3.6 shows the possible faults between different cells of a NAND flash memory.

Resistive shorts between adjacent cells in the same column/row Fig. 3.6.(a) shows the shorts between adjacent cells in the same column. Also the Select Gate (SG) is considered. In particular, they are shorts between FG-FG, FG-SG1, FG-SG2, CG-CG, CG-SG1, CG-SG2.

Fig. 3.6.(b) shows the shorts between adjacent cells in the same row. In particular, they are shorts between FG-FG, D-D, S-S and BL-BL [58].

We will refer at the bridging faults of Fig. 3.6.(a) and Fig. 3.6.(b) as Coupling Fault between Adjacent Cells (CFAC). In particular, Table 3.2 reports the CFAC for each row and each column are reported as CFAC_{row} and CFAC_{col} respectively.

Resistive shorts in the selected transistors Fig. 3.6.(c) shows the shorts in the selected transistors. They are Bitline, Column, Word-line, Select and Gate line opens. It can be shown that the faults of Fig. 3.6.(c) are equivalent to a Stuck-At Fault (SAF) behavior. Therefore, they are reported in Table 3.2 as SAF.

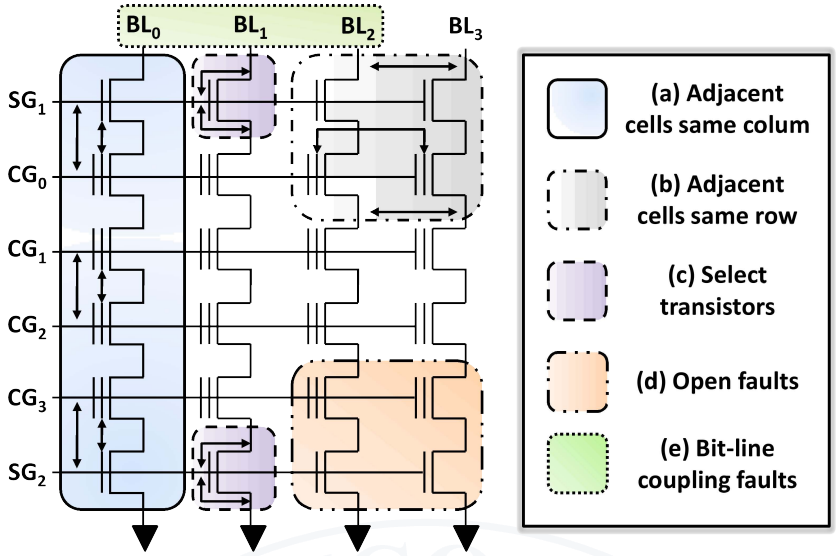


Figure 3.6: NAND Flash memory Inter-cells Faults

Open faults Fig. 3.6.(d) shows the open faults. They are shorts between SG1-D, SG1-S, SG2-D, SG2-CG. It can be shown that the faults of Fig. 3.6.(d) are equivalent to a Stuck-At Fault (SAF) behavior. Therefore, they are reported in Table 3.2 as SAF.

Bit-Line coupling faults The parasitic capacitances connecting the BLs can produce errors; Fig. 3.6.(e) shows how two hypothetical parasitic capacitors connecting BL1 with BL2 and BL2 with BL3 could lead to possible errors when reading cells belonging to the centering column BL2. Table 3.2 refers to this phenomenon as BL Coupling (BC) among three adjacent BLs.

Cell to cell interferences The Capacitive Coupling (CC) faults of [64] are functionally identical to $CFAC_{row}$. The Direct Coupling or Direct field effects (DC) faults of [91] are functionally identical to $CFAC_{col}$. Therefore, they are both already included in our comprehensive fault model.

3.3 A Comprehensive Fault Model for NAND flash

Table 3.1 sums up NAND flash memory disturbances.

Table 3.2 sums up the NAND flash circuit level faults respectively.

Disturbance	Initial state of faulty cell	Fault Excitation	Resulting Error
WPD	C_{ix}='1'	Program any C_{ij} with $j \neq x$	C_{ix}='0'
WED	C_{ix}='0'	Program any C_{ij} with $j \neq x$	C_{ix}='1'
BPD	C_{xj}='0'	Program any C_{ij} with $i \neq x$	C_{xj}='1'
BED	C_{xj}='1'	Program any C_{ij} with $i \neq x$	C_{xj}='0'
RDA(P)	C_{ij}='1'	Read C_{ij} N times	C_{ij}='0'
RDA(E)	C_{ij}='0'	Read C_{ij} N times	C_{ij}='1'
RDU(P)	C_{xj}='1'	Read C_{ij} with $i \neq x$	C_{xj}='0'
RDU(E)	C_{xj}='1'	Read C_{ij} with $i \neq x$	C_{xj}='1'
OED	C_{ij}='1'	Program C_{ij}	C_{ij}='1'
OEP	C_{xj}='0'	Erase any C_{ij} with $i \neq x$	C_{ij}='0'

Table 3.1: NAND Flash Memory Disturbances

Fault	Initial state of faulty cell	Fault Excitation	Resulting Error
SAF0	C_{ij}='0'	Erase C_{ij}	C_{ij}='0'
SAF1	C_{ij}='1'	Program C_{ij}	C_{ij}='1'
$CFAC_{row}$	C_{ij}='1', $C_{i+1,j}$='1'	Program C_{ij}	C_{ij}='1', $C_{i+1,j}$='1' or C_{ij}='0', $C_{i+1,j}$='0'
$CFAC_{col}$	C_{ij}='1', $C_{i,j+1}$='1'	Program C_{ij}	C_{ij}='1', $C_{i,j+1}$='1' or C_{ij}='0', $C_{i,j+1}$='0'
BC	C_{ij}='1', $C_{i,j+1}$='1' $C_{i,j+2}$='1'	Program $C_{i,j+1}$	$C_{i,j+1}$='1'

Table 3.2: NAND Flash Memories Circuit Level Faults

Therefore, Table 3.1 and 3.2 define a comprehensive set of fault models for NAND flash memory. We do not perform any simplification or reduction based on specific technology information for sake of generalization. Therefore, the defined fault model is also technology independent.

3.3.1 The BF&D Extended Test Algorithm

We present a novel test method for NAND Flash memories. It covers the fault models of Tables 3.1 and 3.2. Our test algorithm is an extension of the *Bridging Fault & Disturbances* (BF&D) algorithm proposed in [70]. BF&D was originally developed for NOR flash memories. Two main aspects of the algorithm have been modified: (i) data representation has been adapted in order to have a page oriented algorithm; (ii) the number of read operations has been devised to cover the Read Disturbance discussed in Subsection 3.1.2.

Fig. 3.7 graphically describes the steps of the BF&D Extended algorithm. We suppose to have a block made of m rows and n columns, with $m = n = 4$.

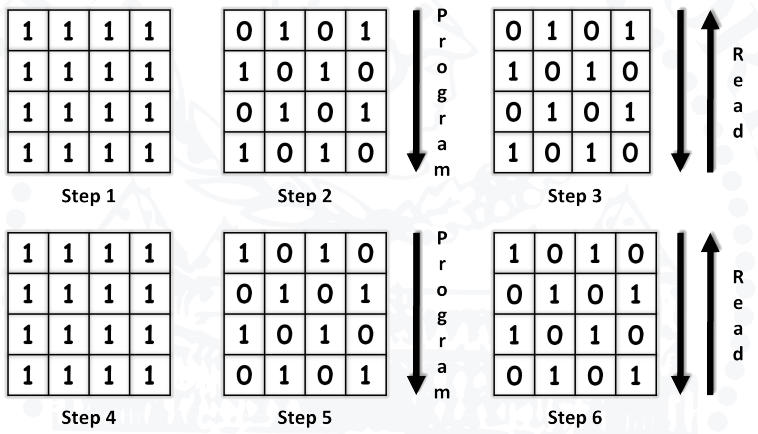


Figure 3.7: BF&D Extended Test Algorithm

Step 1 erases the block of the flash. *Step 2* program each page of the block with a chessboard pattern. *Step 3* performs k read operations on each page of the block. *Step 4* erases the whole block again. *Step 5* program each page of the block with the opposite chessboard pattern of *Step 2*. *Step 6* performs k read operations on each page of the block. k is a selectable parameter of the algorithm[7]. It is strictly linked with technology

[7] for simplicity, Fig. 3.7 has $k = 2$

as discussed in Subsection 3.1.2.

3.3.2 Algorithm Complexity

Eq. 3.1 shows the test time per block.

$$TestBlockTime = (2 \cdot E + 2 \cdot k \cdot n \cdot R + 2 \cdot n \cdot P) \tag{3.1}$$

E is the block erase time, n is the number of pages in a block, R is the page read time, P is the page program time. Eq. 3.2 shows the total time needed to test the NAND flash.

$$TotalTestTime = (N \cdot TestBlockTime) \tag{3.2}$$

N is the number of blocks composing the memory.

3.4 To test or not to test: an important remark

Testing a commercial NAND flash memory can be tough in terms of feasibility and complexity. In fact, the Flash Translation Layer can prevent the tester from accessing to the desired page/block because of the implemented WL/GC transparent strategies. In order to avoid this problem, we should have access to the bare chip or at least to the memory controller and it is not always the case. This is why testing is usually done after production. Vendors have directly access to the bare chip. Therefore, they can easily test and understand the quality of their NAND chips [3].

Flash File System may represent an easier way to test NAND flashes, since there is no FTL and they have direct access to the NAND chip. However, testers have to deal with the so called Memory Technology Device (MTD) layer[8]. Furthermore, FFS potentialities are limited to embedded NAND flash and cannot be extended to commercial ones (e.g., USB sticks).

In conclusion, testing is usually adopted for improving the NAND flash-memory yield, while it is less feasible during the life of the device. Any algorithm aimed at testing the faults of Tables 3.1 and 3.2 is a valid option for screening NAND flash quality. The choice of the algorithm is done accordingly with the (company) requirements in terms of accuracy (e.g., how many faults to test) and complexity (e.g., test time). Finally, after produc-

[8] refer to Chapter 5 for more details

tion, ECC and WL techniques will play a fundamental role for guaranteeing and improving the overall data retention and endurance of the NAND flash device.

SUMMARY

This chapter introduced the main concepts related to flash-memory testing. Testing is commonly adopted to understand the quality of the flash and to improve the yield. However, it is less feasible during the life of the NAND device. At this stage, developers are mainly exploiting ECC and WL techniques. Flash-memory relies on the Floating Gate (FG) technology. However, FG is intrinsically not perfectly reliable and, combined with the rapid technology scaling down, it may lead to problems in terms of *data retention* and *endurance*. Since testing cannot depend on the particular technology adopted, we need a "technology independent" analysis.
Firstly, we analyzed disturbances. They are faulty behavior belonging only to flash-memory and to to the other memories. They do not damage the cells but simply interfere with their content. Each operation (i.e., read, program/write and erase) implies a related possible disturbances (i.e., Read Disturbance, Program Disturbance, Over-Program Disturbance and Over-Erase Disturbance). To complete this approach, we modeled the NAND flash in terms of resistor and capacitors. After this step, we were able to set up a comprehensive fault model which is independent from technology. Finally, we presented a possible test algorithm able to cover all the proposed faults.

To err is human, to really foul things up requires a computer.

Bill Vaughan

CHAPTER 4

ADAPTABLE ERROR CORRECTING CODES DESIGN FOR NAND FLASH MEMORY

Contents of this chapter

4.1 Background and related works
4.2 Optimized Architectures of Programmable Parallel LFSRs [45]
4.3 BCH Code Design Optimization [45]
4.4 Adaptable BCH Encoder [45]
4.5 Adaptable BCH Decoder [45]
4.6 Experimental Results [45]
4.7 A Cross-Layer Approach for New Reliability-Performance Trade-Offs in MLC NAND Flash Memories [142]
4.8 Conclusions [45]

NAND flash memories are a widespread technology for the development of compact, low-power, low-cost and high data throughput mass storage systems for consumer/industrial electronics and mission critical applications. Manufacturers are pushing flash technologies into smaller geometries to further reduce the cost per unit of storage. This includes moving from traditional SLC technologies, able to store a single bit of information, to MLC technologies, storing more than one bit per cell.

The strong transistor miniaturization and the adoption of an increasing number of levels per cell introduce serious issues related to yield, reliability, and endurance [34, 42, 60, 64, 91]. Error correction codes (ECCs) must therefore be systematically applied. ECCs are a cost-efficient technique to detect and correct multiple errors [2]. Flash memories support ECCs by providing spare storage cells dedicated to system management and parity bit storage, while demanding the actual implementation to the application designer [39, 113]. Choosing the correction capability of an ECC is a trade-off between reliability and code complexity. It is therefore a strategic decision in the design of a flash-based storage system. A wrong choice may either overestimate or underestimate the required redundancy, with the risk of missing the target failure rate. In fact, the reliability of a NAND flash memory continuously decreases over time, since program and erase operations are somehow destructive. At the early stage of their life-time, devices have a reduced error-rate compared to intensively used devices [141]. Therefore, designing an ECC system whose correction capability can be modified in-field is an attractive solution to adapt the correction schema to the reliability requirements the flash encounters during its life-time, thus maximizing performance and reliability.

This chapter proposes the hardware implementation of an optimized adaptable Bose - Chaudhuri - Hocquenghem (BCH) codec core for NAND flash memories and a related framework for its automatic generation.

Even though there is a considerable literature about efficient BCH encoder/decoder software implementations [1, 31, 32], modern flash-based memory systems (e.g., Solid State Drives (SSDs)) usually resort to specific high speed hardware IP core [33, 81] in order to minimize the memory latency. This is motivated by the fact that contemporary high-density MLC flash memories require a more powerful error correction capability, and, at the same time, they have to meet more demanding requirements in terms of read/write latency.

Given this premise, we will tackle a BCH hardware implementation for encoding and decoding tasks. In particular, the main contribution of the proposed architecture is its adaptability. It enables in-field selection of the desired correction capability, coupled with high optimization that minimizes the required resources. Experimental results compare the proposed architecture with typical BCH codecs proposed in the literature.

The implementation is supported by the novel ADaptive ECC Automatic GEnerator (ADAGE) design environment . This tool is able to automatically generate, in a para-

metric way, the whole code for each possible architecture. ADAGE concepts will shortly introduced, when required, in the next sections.

The chapter is organized as follows: Section 4.1 shortly introduces basic notions and related works. Sections 4.2 and 4.3 present a solution to reduce resources overhead while Section 4.4 and 4.5 overview the proposed adaptable architecture. Section 4.6 provides experimental results, Section 4.7 overviews a join research with Universita' degli Studi di Ferrara and Section 4.8 summarizes the main contributions of the work and concludes the chapter.

4.1 Background and related works

Several hard- and soft-decision error correction codes have been proposed in the literature, including Hamming based block codes [57, 84], Reed-Solomon codes [108], Bose-Chaudhuri-Hocquenghem (BCH) codes [12], Goppa codes [10], Golay codes [54], etc.

Even though selected classes of codes such as Goppa codes have been demonstrated to provide high correction efficiency [10], when considering the specific application domain of flash memories, the need to trade-off code efficiency, hardware complexity and performances have moved both the scientific and industrial community toward a set of codes that enable very efficient and optimized hardware implementations [33, 71].

Old SLC flash designs used very simple Hamming based block codes. Hamming codes are relatively straightforward and simple to implement in both software and hardware, but they offer very limited correction capability [57, 84]. As the error rate increased with successive generations of both SLC and MLC NAND flash memories, designers moved to more complex and powerful codes including Reed-Solomon (RS) codes [108] and Bose-Chaudhuri-Hocquenghem (BCH) codes [12]. Both codes are similar and belong to the larger class of cyclic codes which have efficient decoding algorithms due to their strict algebraic architecture, and enable very optimized hardware implementations. RS codes perform correction over multi-bit symbols and are better suited when errors are expected to occur in bursts, while BCH codes perform correction over single-bit symbols and better perform when bit errors are not correlated, or randomly distributed. In fact, several studies have reported that NAND flash memories manifest non-correlated or randomly distributed bit errors over a page [138] making BCH codes more suitable for their protection.

An exhaustive analysis of the mathematics governing BCH code is out of the scope of

this chapter. Only those concepts required to understand the proposed hardware implementation will be shortly discussed. It is worth to mention here that, since several publications proposed very efficient hardware implementations of Galois fields polynomial manipulations, such manipulation will be used in both encoding and decoding operations [73, 81, 108].

Given a finite Galois field $GF(2^m)$ (with $m \geq 3$), a t-error-correcting BCH code, denoted as $BCH[n,k,t]$, encodes a k-bit message $b_{k-1}b_{k-2}\ldots b_0$ ($b_i \in GF(2)$) to a n-bit codeword $b_{k-1}b_{k-2}\ldots b_0\, p_{r-1}p_{r-2}\ldots p_0$ ($b_i, p_i \in GF(2)$) by adding r parity bits to the original message. The number r of parity bits required to correct t errors in the n-bit codeword is computed by finding the minimum m that solves the inequality $k + r \leq 2^m - 1$, where $r = m \cdot t$. Whenever $n = k + r < 2^m - 1$, the BCH code is called *shortened* or *polynomial*. In a shortened BCH code the codeword includes less binary symbols than the ones the selected Galois field would allow. The missing information symbols are imagined to be at the beginning of the codeword and are considered to be 0. Let α be a primitive element of $GF(2^m)$ and $\psi_1(x)$ a primitive polynomial with α as a root. Starting from $\psi_1(x)$ a set of minimal polynomials $\psi_i(x)$ having α^i as root can be always constructed [98]. For the same $GF(2^m)$, different valid $\psi_1(x)$ may exist [131]. The generator polynomial $g(x)$ of a t-error-correcting BCH code is computed as the Least Common Multiple (LCM) among $2t$ minimal polynomials $\psi_i(x)$ ($1 \leq i \leq 2t$). Given that $\psi_i(x) = \psi_{2i}(x)$ ($\forall i \in [1,t]$) [2], only t minimal polynomials must be considered and $g(x)$ can therefore be computed as:

$$g(x) = LCM\left[\psi_1(x), \psi_3(x) \ldots, \psi_{2t-1}(x)\right] \tag{4.1}$$

When working with BCH codes, the message and the codeword can be represented as two polynomials: (1) $b(x)$ of degree $k-1$ and (2) $c(x)$ of degree $n-1$. Given this representation, both the encoding and the decoding process can be defined by algebraic operations among polynomials in $GF(2^m)$. The encoding process can be expressed as:

$$c(x) = m(x) \cdot x^r + Rem\left(m(x) \cdot x^r\right)_{g(x)} \tag{4.2}$$

where $Rem(m(x) \cdot x^r)_{g(x)}$ denotes the remainder of the division between the message left shifted of r positions and the generator polynomial $g(x)$. This remainder represents the r parity bits to append to the original message.

The BCH decoding process searches for the position of erroneous bits in the codeword. This operation requires three main computational steps: 1) syndrome computation, 2) error locator polynomial computation, and 3) error position computation.

Given the selected correction capability t, the decoding process requires first the computation of $2t$ syndromes of the codeword $c(x)$, each associated with one of the $2t$ minimal polynomials $\psi_i(x)$ generating the code. Syndromes are calculated by first computing the remainders $R_i(x)$ of the division between $c(x)$ and each minimal polynomial $\psi_i(x)$. If all remainders are null, $c(x)$ does not contain any error and the decoding stops. Otherwise, the $2t$ syndromes are computed by evaluating each remainder $R_i(x)$ in α^i: $S_i = R_i\left(\alpha^i\right)$. Practically, according to (4.1), given that $\psi_i(x) = \psi_{2i}(x)$, only t remainders must be computed and evaluated in $2t$ elements of $GF(2^m)$.

The most used algebraic method to compute the coefficients of the error locator polynomial from the syndromes is the Berlekamp-Massey algorithm [11]. Since the complexity of this algorithm grows linearly with the correction capability of the code, it enables efficient hardware implementations. The equations that link syndromes and error locator polynomial can be expressed as:

$$\begin{pmatrix} S_{t+1} \\ S_{t+2} \\ \vdots \\ S_{2t} \end{pmatrix} = \begin{pmatrix} S_1 & S_2 & \dots & S_t \\ S_2 & S_3 & \dots & S_{t+1} \\ \vdots & \vdots & & \vdots \\ S_t & S_{t+1} & \dots & S_{2t-1} \end{pmatrix} \cdot \begin{pmatrix} \lambda_t \\ \lambda_{t-2} \\ \vdots \\ \lambda_0 \end{pmatrix} \tag{4.3}$$

The Berlekamp-Massey algorithm iteratively solves the system of equations defined in (4.3) using consecutive approximations.

Finally, the Chien Machine searches for the roots of the error locator polynomial $\lambda(x)$ computed by the Berlekamp-Massey algorithm [29]. It basically evaluates the polynomial $\lambda(x)$ in each element α^i of $GF(2^m)$. If α^i satisfies the equation $1+\lambda_1\alpha^i+\lambda_2\alpha^{2i}+\ldots+\lambda_t\left(\alpha^i\right)^t = 0$, α^i is a root of the error locator polynomial $\lambda(x)$, and its reciprocal 2^m-1-i reveals the error position. In practice, this computation is performed exploiting the iterative relation:

$$\lambda\left(\alpha^{j+1}\right) = \lambda_0 + \sum_{k=1}^{t-1}\left[\lambda_k\left(\alpha^j\right)^k\right]\alpha^k \tag{4.4}$$

Several publications proposed optimized hardware implementations of BCH codecs with fixed correction capability [33, 56, 71, 82, 107, 127]. However, to the best of our

knowledge, only Chen et al. proposed a solution allowing limited adaptation by extending a standard BCH codec implementation [25]. One of the main contributions of Chen et al. is a Programmable Parallel Linear Feedback Shift Register (PPLFSR), whose generic architecture is reported in Fig. 4.1. It enables to dynamically change the generator polynomial of the LFSR. This is a key feature in the implementation of an adaptable BCH encoder.

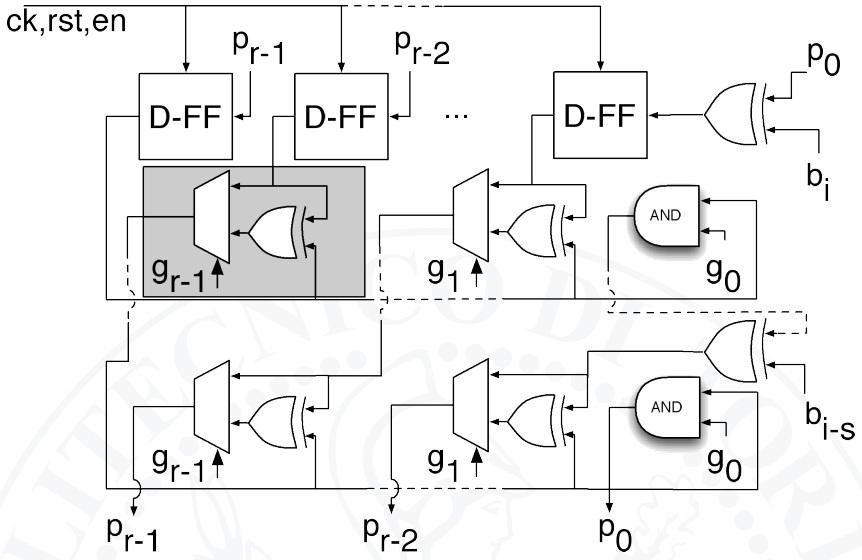


Figure 4.1: Architecture of a r-bit PPLFSR with s-bit parallelism.

The gray box of Fig. 4.1 highlights the basic adaptable block of this circuit. It exploits a multiplexer, controlled by one of the coefficients of the desired divisor polynomial, to dynamically insert an XOR gate at the output of one of the related D-type flip-flops composing the register. The s vertical stages of the circuit implement the parallelism of the PPLFSR computing the state at clock cycle $i+s$, based on the state at cycle i. However, this solution has high overhead. In fact such PPLFSR is able to divide by all possible r-bit polynomials, while just well selected divisor polynomials are required.

Although Chen at al. deeply analyze the encoding process and the issues related to the storage of parity bits, the decoding process is scarcely analyzed, without providing details on how adaptability is achieved. Four different correction modes, namely $t = (9, 14, 19, 24)$ are considered in [25] for a BCH code defined on $GF(2^{13})$ with a block size of 512B (every 2KB page of the flash is split in four blocks). The selection of the 4 modes is based on considerations about the number of parity bits to store. How-

ever, there is no provision to understand whether additional modes can be easily implemented. As an example, when selecting correction modes in which the size of the codeword is not a multiple of the parallelism of the decoder, alignment problems arise, which are completely neglected in the paper.

4.2 Optimized Architectures of Programmable Parallel LFSRs

In this section, we will introduce an optimized block to perform an adaptable remainder computation. In fact, one of the most recurring operations in BCH encoding/decoding is the remainder computation between a polynomial representing a message to encode/decode and a generator/minimal polynomial of the code, that depends on the selected correction capability. The PPLFSR of Fig. 4.1 can perform this operation [25].

A r-bit PPLFSR can potentially divide by any r-bit polynomial by properly controlling its configuration signals ($g_0 \dots g_{r-1}$). However, in BCH encoding/decoding, even considering an adaptable codec, just well selected divisor polynomials are required (e.g., the generators polynomials $g_9(x)$, $g_{14}(x)$, $g_{19}(x)$, $g_{24}(x)$ of the four implemented correction modes of [25]). This computational block is therefore highly inefficient. Moreover, the set of divisor polynomials required in a BCH codec usually share common terms among each other. Such terms can be exploited to generate an optimized PPLFSR (OPPLFSR) architecture.

Let us consider, as an example, the design of a r=15-bit programmable LFSR able to divide by two polynomials $p_1(x) = x^{15} + x^{13} + x^{10} + x^5 + x^3 + x + 1$ and $p_2(x) = x^{13} + x^{12} + x^{10} + x^5 + x^4 + x^3 + x^2 + x + 1$ using a s=8-bit parallelism.

A traditional PPFLSR implementation would require $15 \times 8 = 120$ gray boxes (i.e., 120 XORs-MUXs). According to this implementation, this PPLFSR could divide by any $2^{15} = 32,768$ possible 15-bit polynomials, even if just 2 polynomials (i.e., the 0.006% of its full potential) are required.

An analysis of the target divisor polynomials can be exploited to optimize the PPLFSR architecture. Table 4.1 reports the binary representation of the two polynomials.

Looking at Table 4.1, three categories of polynomial terms can be identified:

1. Common terms (represented in bold), i.e., terms defined in all considered polynomials (x^{13}, x^{10}, x^5, x^3, x, and 1 in Table 4.1). For these terms, an XOR will be always

Table 4.1: An example of the representation of $p_1(x)$ and $p_2(x)$

	x^{15}	x^{14}	x^{13}	x^{12}	x^{11}	x^{10}	x^9	x^8	x^7	x^6	x^5	x^4	x^3	x^2	x^1	1
$p_1(x)$	1	$\underline{0}$	**1**	0	$\underline{0}$	**1**	$\underline{0}$	$\underline{0}$	$\underline{0}$	**1**	0	**1**	0	**1**	**1**	**1**
$p_2(x)$	0	$\underline{0}$	**1**	1	$\underline{0}$	**1**	$\underline{0}$	$\underline{0}$	$\underline{0}$	**1**	1	**1**	1	**1**	**1**	**1**

required in the PPLFSR, thus saving the area dedicated to the MUX and the related control logic.

2. Missing terms (represented in underlined italic zeros), i.e., terms not defined in any of the considered polynomials, (x^{14}, x^{11}, x^9, x^8, x^7 and x^6 in Table 4.1). For these terms both the XOR and the related MUX can be avoided.

3. Specific terms, i.e., terms that are specific of a subset of the considered polynomials (x^{15}, x^{12}, x^4, x^2 in Table 4.1). These terms are the only ones actually required.

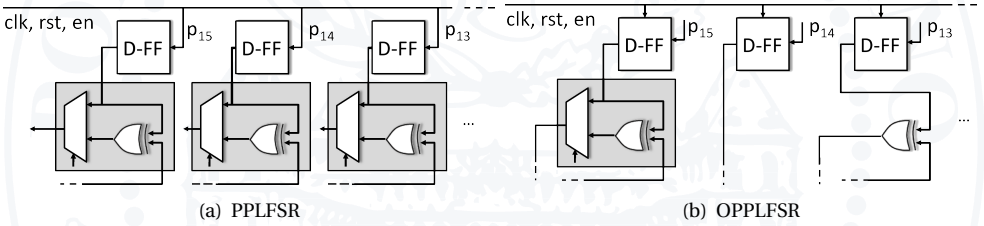
We can therefore implement an optimized programmable LFSR (OPPLFSR) with three main building blocks:

1. each common present term (i.e., columns of all "1" of Table 4.1) needs an XOR, only;

2. each common absent term (i.e., columns of all "0" of Table 4.1) needs neither XOR nor MUX;

3. each specific term has a gray box, as Fig. 4.1;

Fig. 4.2 shows the resulting design for the portion x^{15}, x^{14} and x^{13}.



(a) PPLFSR (b) OPPLFSR

Figure 4.2: Example of the resulting PPLFSR (a) and OPPLFSR (b) with 8-bit parallelism for x^{15}, x^{14} and x^{13} of $p_1(x)$ and $p_2(x)$

This optimization also applies on polynomials with very different lengths. As an example, an OPPLFSR with single bit parallelism and able to divide by $p_1(x) = x^{225} + x + 1$ and $p_2(x) = x + 1$, would only require a single adaptable block, compared to the 226 blocks required by a normal PPLFSR. Furthermore, the advantage of the OPPLFSR increases with the parallelism of the block. In fact, with the same 2 polynomials, a 8-bit OPPLFSR would require 8 adaptable blocks compared to $226 \times 8 = 1,808$ adaptable blocks of a traditional PPLFSR.

For sake of generality, Fig. 4.3 shows the high-level architecture of a generic OPPLFSR. Such a block is able to divide by a set $p_1(x), ..., p_M(x)$ of polynomials. We denote with q the number of required gray boxes.

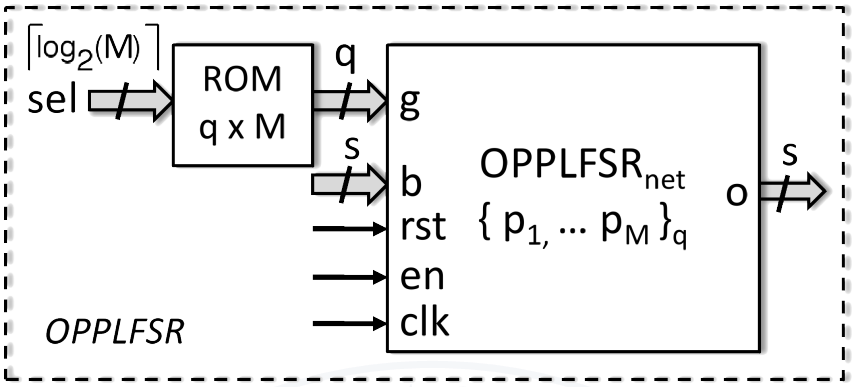


Figure 4.3: High-level architecture of the OPPLFSR

The OPPLFSR interface includes: a s-bit input port (`b`) used to feed the data, a $\lceil \log_2(M) \rceil$-bit input port (`sel`) used to select the polynomial of the division, and a s-bit port (`o`) providing the result of the division. Two blocks compose the OPPLFSR: $OPPLFSR_{net}$ and *ROM*. The $OPPLFSR_{net}$ represents the complete network, partially shown in the example of Fig. 4.2. Given the output of the ROM, the q-bit signal g controls the MUXs of the q gray boxes (Fig. 4.2) according to the selected polynomial. The ROM is optimized accordingly with the design of the OPPLFSR, which leads to a reduced ROM and to a lower area overhead w.r.t. a full PPLFSR.

4.3 BCH Code Design Optimization

In this section, we address first the issue of choosing the most suitable set of polynomials for an optimized adaptable BCH code. Then, we propose a novel block, shared between the adaptable BCH encoder and the decoder, which reduces the area overhead of the resulting codec core.

4.3.1 The choice of the set of polynomials

The optimization offered by the OPPLFSR introduced in Section 4.2, may become ineffective if not properly exploited. It depends on the number and on the terms of the shared divisor polynomials implemented in the block. As an example, an excessive num-

ber of shared polynomials may make it difficult to find common terms, leading to an unwilled increase of the area overhead. Therefore, the choice of the polynomials to share is critical and must be properly tailored to the overall design.

Let us denote by Ω the set of t generators $g_i(x)$ and t minimal polynomials ψ_i which fully characterize an adaptable BCH code (see Section 4.1). Since for $GF(2^m)$ several primitive polynomials $\psi_i(x)$ can be used to define the code, several set Ω_i can be constructed. Choosing the most suitable set Ω_i is critical to obtain an effective design of the OPPLFSR. On the one hand, it can be shown that the complexity of Ω_i increases with m [79, 98, 131]. On the other hand, the current trend is to adopt BCH codes with high values of m (e.g., $GF(2^{15})$) because current flash devices features a worse bit error rate [42]. Therefore, a simple visual inspection of each set Ω_i is not feasible to find the most suitable set of polynomials. An algorithmic approach is therefore mandatory.

Each set Ω_i can be classified resorting to a *Maximum Correlation Index* (MCI). We define as $MCI(p_1, p_2, ..., p_N)$ the maximum number of common terms shared by a generic set of polynomials $p_1, p_2, ..., p_N$. As an example, the polynomials of Table 4.1 have $MCI(p_1, p_2) = 12$.

In the sequel, we introduce an algorithm to assess each set Ω_i according to its MCI. Given $i = \{1, ..., Y\}$, for each set Ω_i:

1. consider $\Omega_i = \{p_1, ..., p_N\}$ and $v_0 = p_1$;
2. determine the polynomial p_h such that the partition $S_{i,1} = (v_0, p_h)$ has the maximum $MCI(v_0, p_h)$, where $h = \{1, ..., N\}$ and $p_h \neq v_0$;
3. determine the polynomial p_k such that the partition $S_{i,1} = ((v_0, p_h), p_k)$ has the maximum $MCI(v_0, p_h, p_k)$, where $k = \{1, ..., N\}$ and $p_k \neq p_h \neq v_0$;
4. repeat step 3 until all polynomials have been considered in the partition $S_{i,1}$;
5. change the starting polynomial to the next one, e.g., $v_0 = p_2$, considering $S_{i,2}$ and repeat steps 2-4;
6. when $v_0 = p_N$, consider the next set Ω_{i+1};

The algorithm ends when all sets Ω_i have been analyzed. For each Ω_i, the output is a set of partitions:

$$S_{i,j} = \{S_{i,1}, S_{i,2}, ..., S_{i,N}\} \tag{4.5}$$

Fig. 4.4 graphically shows the MCI of two partitions generated from two different starting points, for an hypothetical set Ω_i.

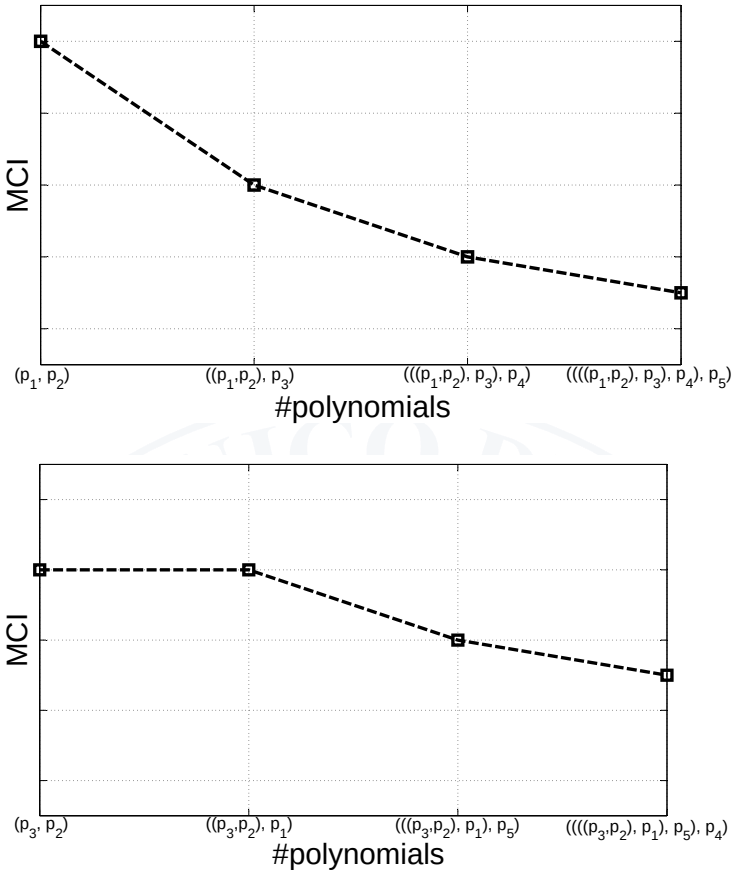


Figure 4.4: MCI examples of two hypothetical partitions $S_{i,1}$ and $S_{i,2}$

Fig. 4.4 shows that MCI always has a decreasing trend with the size of the partition S. This is straightforward since adding a polynomial may only decrease or keep constant the current value of MCI. The curves, reported in 4.4, are critical in the choice of the most suitable set of polynomials for an optimized BCH code. For each partition $S_{i,j}$ with $j = \{1...N\}$, we can compute the average MCI (MCI_{avg}) as:

$$MCI_{avg}(S_{i,j}) = \frac{1}{N}\sum_{l=1}^{N-1} MCI_l \tag{4.6}$$

Eq. 4.6 applies to each set Ω_i where $i = \{1...Y\}$.

The best partition of the set Ω_i is then computed selecting the one with maximum MCI_{avg}:

$$S_{best_i} = \underset{j}{argmax}\left[MCI_{avg}\left(S_{i,j}\right)\right] \tag{4.7}$$

Finally, Eq. 4.8 compares the best partition of each set Ω_i to find the best set of polynomials:

$$S_{bestBCH} = \underset{i}{argmax}\left[S_{best_i}\right] \tag{4.8}$$

Eq. 4.8 defines the family of polynomials $S_{bestBCH}$, with the maximum average number of common terms.

Table 4.2: An example of Ω_i

	x^6	x^5	x^4	x^3	x^2	x^1	1
p_1	1	0	1	0	0	1	0
p_2	1	1	0	1	0	1	1
p_3	1	0	1	1	1	1	1
p_4	0	1	1	0	0	0	1
p_5	1	1	0	1	1	0	1
p_6	0	0	1	0	0	1	1

Let us provide an example to support the understanding of the algorithm. Suppose to consider a single set Ω_i composed of the polynomials of Table 4.2. The steps of the algorithm are:

1. Let us start with $v_0 = p_1$
2. We first evaluates $MCI\left(p_1, p_2\right) = 3$, $MCI\left(p_1, p_3\right) = 4$, $MCI\left(p_1, p_4\right) = 3$. Since $MCI\left(p_1, p_3\right) = 4$ is the maximum, the resulting partition is $S_{i,1} = \left\{p_1, p_3\right\}$
3. The next step considers $MCI(\left(p_1, p_3\right),\ p_2) = 3$ and $MCI(\left(p_1, p_3\right),\ p_4) = 3$. It is straightforward that the choice of either p_2 or p_4 does not affect the final value of the MCI_{avg}.

Given Ω_i with starting point p_1, it can be shown that the final partition is $S_{i,1} = \left\{\left(\left(p_1, p_3\right), p_4\right), p_2\right\}$ with a $MCI_{avg} = {}^{(4+3+3)}/_4 = 2.5$ from Eq. 4.6.

The complete algorithm iterates this computation for all possible starting points. Fig. 4.5 graphically shows the output of the MCI associated with each partition $S_{i,j}$ calculated for the following starting point $j = \{1,2,3,4\}$.

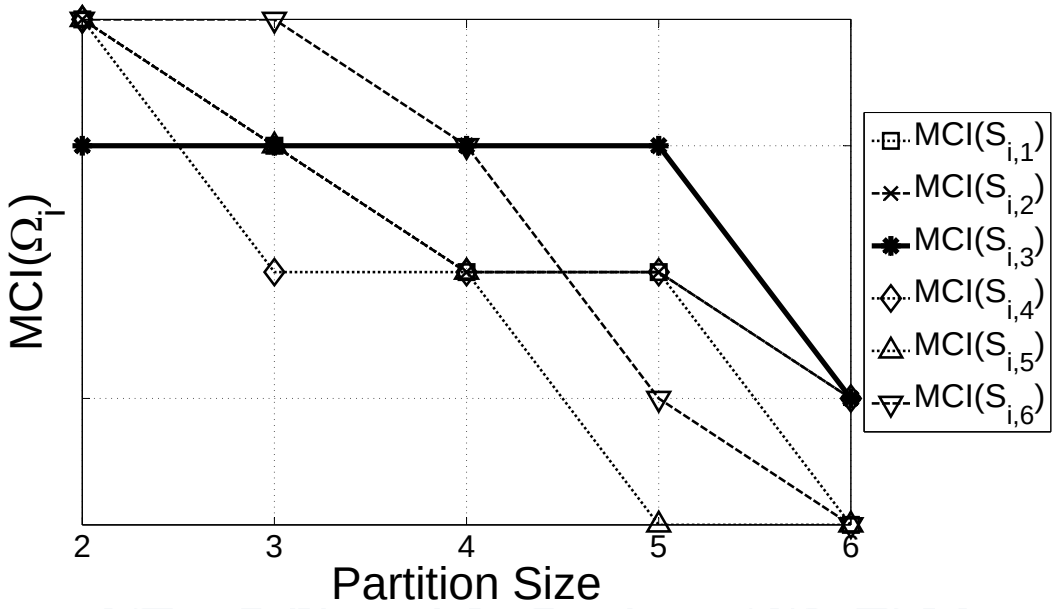


Figure 4.5: The MCI Trend of Table 4.2

According to Eq. 4.7, $S_{i,2}$ (the bold line) is the S_{best_i} of the example of Table 4.2, with a $MCI_{avg}\left(S_{i,j}\right) = 4$.

4.3.2 Shared Optimized Programmable Parallel LFSRs

Let us assume to design an adaptable BCH code with correction capability from 1 up to t_M. Such a code needs to compute remainders of the division of:

- the message $m(x)$ by (potentially) all generator polynomials from g_1 up to g_{t_M}, for the encoding (4.2);
- the codeword $c(x)$ by (potentially) all minimal polynomials from $\psi_1(x)$ up to $\psi_{2t_M-1}(x)$, to compute the set of syndromes required during the decoding phase.

In a traditional implementation, these computations are performed by two separate set of LFSRs. In this chapter, we propose to devise a shared set of LFSRs able to: (i) perform all these computations, and (ii) reduce the overall cost in terms of resources

overhead. Therefore, we can adopt the same shared set of LFSRs both in the encoding and decoding processes. This is possible since in a flash memory these operations are, in general, not required at the same time.

The OPPLFSR, introduced in Section 4.2, is the main building block of the set of shared LFSRs. Therefore, we will refer hereafter to such set of LFSRs as shared OPPLFSR (shOPPLFSR). Fig. 4.6 shows the high-level architecture of the shOPPLFSR. Its interface includes: a s-bit input port (`IN`) used to input the data to be divided, a $\lceil \log_2(N) \rceil$-bit input port (`en`) used to enable each OPPLFSR, an input port (`sel`) used to select the proper polynomial by which each OPPLFSR has to divide, and a N × s-bit port (`p`) providing the result of the division.

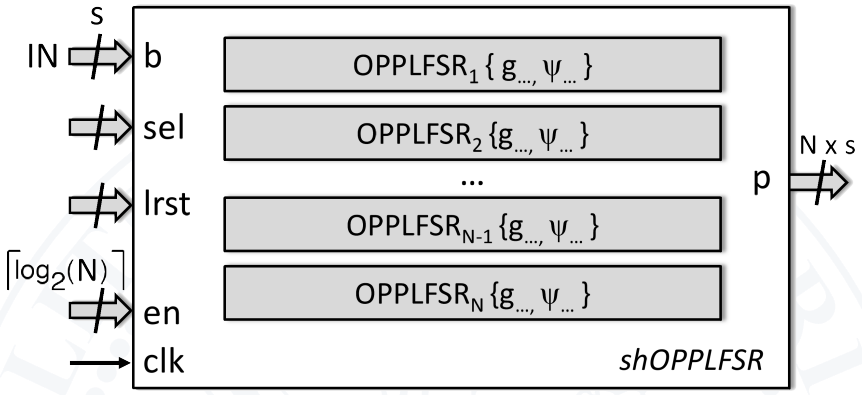


Figure 4.6: The shOPPLFSR architecture is composed by multiple OPPLFSRs

Given N OPPLFSRs and a maximum correction capability t_M, each OPPLFSR_i performs the division by a set of generator polynomials $g(x)$ and minimal polynomials $\psi(x)$. Such shOPPLFSR can be seen as an optimized programmable LFSR able to:

- divide by all generator polynomials from $g_1(x)$ to $g_{t_M}(x)$;
- divide by specific subsets of minimal polynomials from Eq. 4.1, as well.

An improper choice of the shared polynomials $g(x)$ and $\psi(x)$ can dramatically reduce the performance of the overall BCH codec. Also the partitioning strategy adopted is critical to maximize the optimization in terms of area, minimizing the impact on the latency of encoding/decoding operations.

The algorithm presented in Section 4.3.1 provides a valuable support for the exploration of this huge design space. In fact, the proposed method can be exploited to prop-

erly partition polynomials into the different OPPFLSRs of Fig. 4.6, in order to maximize the optimization of the resulting shOPPFLSR. Such optimization should not be obtained following blindly the outcomes of the algorithm, but always tailoring them to the specific design. Regarding this topic, Section 4.6 provides more details about our experimental setup and the related experimental results.

4.4 Adaptable BCH Encoder

In this section, we propose an adaptable BCH encoder which exploits the shOPPLFSR of Section 4.3. According to the BCH theory, the shOPPLFSR of Fig. 4.6 is a very efficient circuit to perform the computation expressed in Eq. 4.2. However, in the encoding phase, the message $m(x)$ must be multiplied by x^r before calculating the reminder of the division by $g(x)$ (see Eq. 4.2). This can be obtained without significant modifications of the architecture of shOPPFLSR. It is enough to input the bits of the message directly in the most significant bit of the LFSR, instead than starting from least significant bit. Fig. 4.7 shows the high-level architecture of the adaptable encoder.

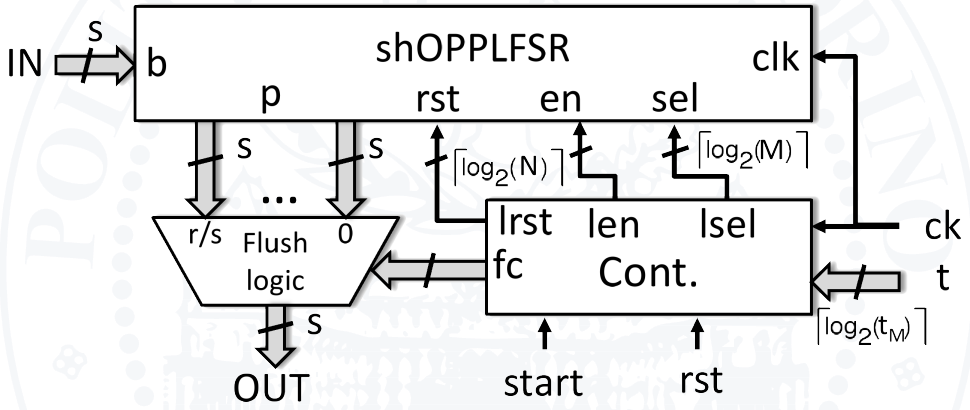


Figure 4.7: High-level architecture of the adaptable encoder highlighting the three main building blocks and their main connections.

The encoder's interface includes: a s-bit input port (`IN`) used to input the k-bit message to encode starting from the most significant bits, a $\lceil \log_2(t_M) \rceil$-bit input port (`t`) selecting the requested correction capability in a range between 1 and t_M, a `start` input signal used to start the encoding process and a s-bit output port (`OUT`) providing the r parity bits. Three blocks compose the encoder: a *shOPPLFSR*, a *flush logic* and a *controller*.

The shOPPLFSR performs the actual parity bits computation. According to the BCH theory, adaptation is achieved by supporting the computation of remainders with t_M generator polynomials, one for each value t may assume. The controller achieves this task in two steps: (i) enabling the proper OPPLFSR through the `len` signal, and (ii) selecting the proper polynomial through the `lsel` signal, according to the desired correction capability `t`. Then, it manages the overall encoding process based on two internal parameters: 1) the number of s-bit words composing the message (fixed at design time) and 2) the number of produced s-bit parity words, that depends on the selected correction capability. The flush logic splits the r parity bits into s-bit words, providing them in output, one per clock cycle.

To further optimize the encoding and the decoding process, since in a flash memory these operations are not required at the same time, the encoder's shOPPLFSR can be merged with the shOPPLFSRs that will be employed in the syndrome computation (see Section 4.5.1), thus allowing additional area saving.

4.5 Adaptable BCH Decoder

Fig. 4.8 presents the high-level architecture of the proposed adaptable decoder. The decoder's interface includes: a $s-$bit input port (`IN`) used to input the $n-$bit codeword to decode (starting from the most significant bits), a $\lceil \log_2(t_M) \rceil -$bit input port (`t`) to select the desired correction capability, a `start` input signal to start the decoding and a set of output ports providing information about detected errors. In particular:

- `deterr` is a $\lceil \log_2(t_M) \rceil -$bit port providing the number of errors that have been detected in a codeword. In case of decoding failure it is set to 0;

- `erradd` and `errmask` provide information about the detected error positions. Assuming the codeword split into $h-$bit words, `erradd` is used as a word address in the codeword and `errmask` is a $h-$bit mask whose asserted bits indicate detected erroneous bits in the addressed word. The parallelism h of the error mask depends on the parallelism of the Chien machine, as explained later in this section;

- `vmask` is asserted whenever a valid error mask is available at the output of the decoder;

- `fail` is asserted whenever an error occurred during the decoding process (e.g., the number of errors is greater than the selected correction capability);
- `end` is asserted when the decoding process is completed.

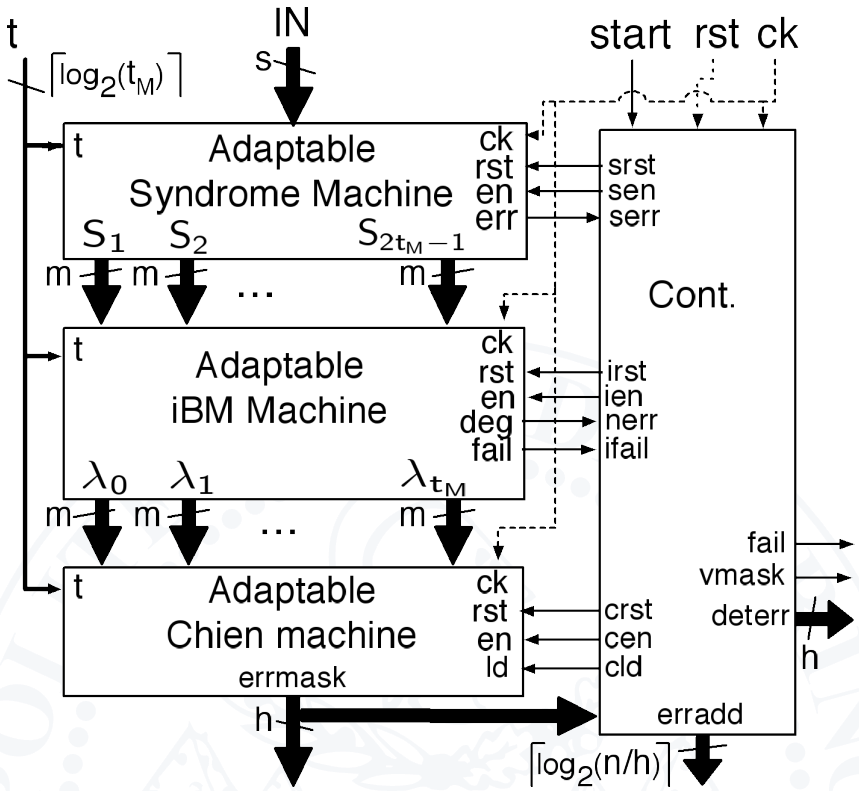


Figure 4.8: High-level architecture of the adaptable decoder, highlighting the four main building blocks: the adaptable syndrome machine, the adaptable iBM machine, the adaptable Chien machine, and the controller in charge of managing the overall decoding process

The full decoder therefore includes four main blocks: (1) the *Adaptable Syndrome Machine*, computing the syndromes of the codeword, (2) the *Adaptable inversion-less Berlekamp Massey (iBM) Machine*, that elaborates the syndromes to produce the error locator polynomial, (3) the *Adaptable Chien Search Machine* in charge of searching for the error positions, and (4) the *Controller* coordinating the overall decoding process.

4.5.1 Adaptable Syndrome Machine

Fig. 4.9 shows the high-level architecture of the proposed adaptable syndrome machine with correction capability $1 \leqslant t \leqslant t_M$.

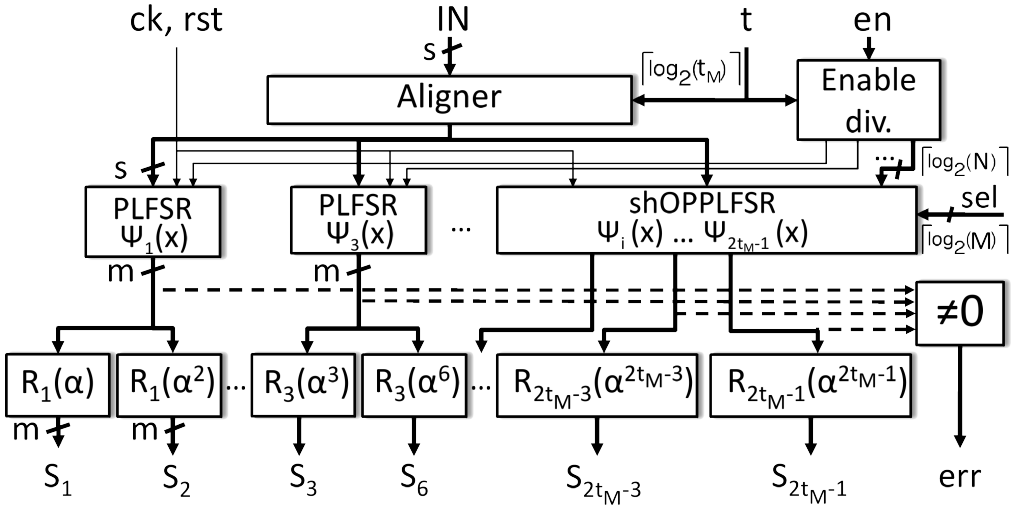


Figure 4.9: Architecture of the adaptable Syndrome Machine

According to Section 4.2, remainders can be calculated by a set of Parallel LFSRs (PLFSRs) whose architecture is similar to the one of the PPLFSR of Fig. 4.1, with the only difference that the characteristic polynomial is fixed (XOR gates are inserted only where needed, without multiplexers). Each PLFSR computes the remainder of the division of the codeword by a different minimal polynomial $\psi_i(x)$. Given two correction capabilities t_1 and t_2 with $t_1 < t_2 \leq t_M$, the set of $2t_1$ minimal polynomials generating the code for t_1 is a subset of those generating the code for t_2. To obtain adaptability of the correction capability in a range between 1 and t_M, the syndrome machine can therefore be designed to compute the maximum number t_M of remainders required to obtain $2t_M$ syndromes. Based on the selected correction capability t, only the first t PLFSRs out of the t_M available in the circuit are actually enabled through the *Enable div.* network of Fig. 4.9.

A full parallel syndrome calculator, including t_M PLFSRs, requires a considerable amount of resources that are underutilized in the early stages of the flash lifetime when reduced correction capability is required. To optimize the adaptable syndrome machine and to trade-off between complexity and performance, we exploit the shOPPLFSR introduced in Section 4.2. The architecture proposed in Fig. 4.9 includes two sets of LFSRs for remainder computation: (i) conventional PLFSRs, and (ii) shOPPLFSR. Conventional PLFSRs are exploited for parallel fast computation of low order syndromes required when the requested correction capability is below a given threshold. shOPPLFSR

is designed to divide for selected groups of minimal polynomials not covered by the fixed PPLFSRs. It represents a shared resource utilized when the requested correction capability increases. It enables area reduction at the cost of a certain time overhead. The architectural design, chosen for the fixed PLFSRs and the OPPLFSR, enables to trade-off hardware complexity and decoding time, as it will be discussed in Section 4.6.

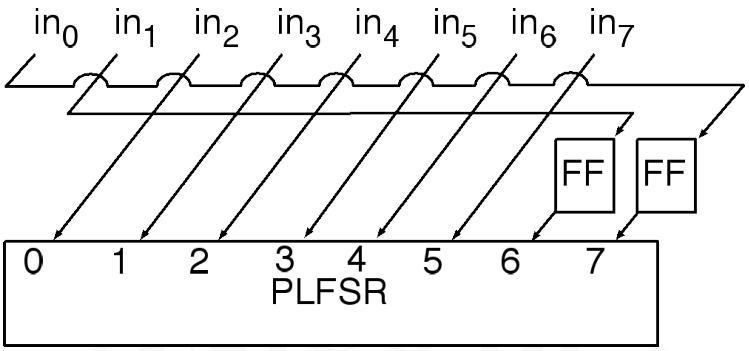


Figure 4.10: Example of the schema of a byte aligner for $t = 2$ and $s = 8$

It is worth to mention here that the parallel architecture of the PLFSR, coupled with the adaptability of the code, introduces a set of additional word alignment problems that must be addressed to correctly adapt the syndrome calculation to different values of t. The syndrome machine receives the codeword in words of s bits, starting from the most significant word. When the number of parity bits does not allow to align the codeword to the parallelism s, the unused bits of the last word are filled with 0. To correctly compute each syndrome, the parity bit r_0 of the codeword must enter the least significant bit of each LFSR. The aligner block of Fig. 4.9 assures this condition by properly right-shifting the codeword while it is input into the syndrome machine. Let us consider the following example: $k = 2$KB, $m = 15$, $t = 2$, $s = 8$ and therefore $r = m \cdot t = 30$. Since 30 is not multiple of $s = 8$, the codeword is filled with two zeros and $\mathtt{p}_0$ is saved in position 2 of the last byte of the codeword ($\mathtt{m}_{2047}\,\mathtt{m}_{2046}...\mathtt{m}_1\,\mathtt{m}_0\,\mathtt{p}_{29}\,\mathtt{p}_{28}...\mathtt{p}_1\,\mathtt{p}_0\,0\,0$). In this case the PLFSRs require a 2-bit alignment, implemented by the network of Fig. 4.10. It simply delays the last 2 input bits resorting to two flip-flops, whose initial state has to be zero, and properly rotates the remaining input bits. Changing the correction capability of the decoder changes the number of parity bits of the codeword, and therefore the required alignment. Given the parallelism s of the decoder, a maximum of s alignments must be provided and implemented in the *Aligner* block of Fig. 4.9.

With the proper alignment, the PLFSRs can perform the correct division and the evaluators can provide the required syndromes. The evaluators are simple combinational networks involving XOR operations, according to the Galois Fields theory (the reader may refer to [81] for specific implementation details).

4.5.2 Adaptable Berlekamp Massey Machine

In our adaptable codec we implemented the inversion-less Berlekamp-Massey (iBM) algorithm proposed in [140] which is able to compute the error locator polynomial $\lambda(x)$ in t iterations.

The main steps of the computation are reported in Alg. 1. At iteration i (rows 2 to 12), the algorithm finds an error locator polynomial $\lambda(x)$ whose coefficients solve the first i equations of (4.3) (row 4). It then tests if the same polynomial solves also $i+1$ equations (row 5). If not, it computes a discrepancy term δ so that $\lambda(x)+\delta$ solves the first $i+1$ equations (row 9). This iterative process is repeated until all equations are solved. If, at the end of the iterations, the computed polynomial has a degree lower than t, it correctly represents the error locator polynomial and its degree represents the number of detected errors; otherwise, the code is unable to correct the given codeword.

1: $\lambda(x)=1$, $k(x)=1$, $\delta=1$
2: **for** i = 0 to t − 1 **do**
3: $d=\sum_{j=1}^{t}\left(\lambda_j \cdot S_{2i-j}\right)$
4: $\lambda(x)=\delta\lambda(x)+d\cdot x\cdot k(x)$
5: **if** $d=0$ OR $Deg(\lambda(x))>i$ **then**
6: $k(x)=x^2\cdot k(x)$
7: **else**
8: $k(x)=x\cdot k(x)$
9: $\delta=d$
10: **end if**
11: i=i+1
12: **end for**
13: **if** $Deg(\lambda(x))<t$ **then**
14: output $\lambda(x)$, $Deg(\lambda(x))$
15: **else**
16: output FAILURE
17: **end if**

Algorithm 1 Inversion-less Berlekamp-Massey alg.

The architecture of the iBM machine is intrinsically adaptive as long as one guarantees that the internal buffers and the hardware structures are sized to deal with the worst case design (i.e., $t = t_M$). The coefficients of $\lambda(x)$ are $m-$bit registers whose number

depends on the correction capability. In the worst case, up to t_M coefficients must be stored for each polynomial.

The adaptable iBM machine therefore includes two $m-$bit register files with t_M registers to store these coefficients. Whenever the requested correction capability is lower than t_M some of the registers will remain unused. The number of multiplications performed during the computations also depends on t. Row 3 requires t multiplications, while row 4 requires t multiplications to compute $\delta\lambda_i(x)$ and t multiplications to compute $d \cdot x \cdot k(x)$.

We implemented a serial iBM Machine including 3 multipliers for GF(2^m) to perform multiplications of rows 3 and 4. It can perform each iteration of the iBM algorithm in $2t$ clock cycles (t cycles for row 3 and t cycles for row 4) achieving a time complexity of $2t^2$ clock cycles. This implementation is a good compromise between performance and hardware complexity. An input t dynamically sets the number of iterations of the algorithm, thus implementing the adaptation.

4.5.3 Adaptable Chien Machine

The overall architecture of the proposed adaptable Chien Machine is shown in the Fig. 4.11. The machine first loads into t_M m-bit registers the coefficients from λ_1 to λ_{t_M} of the error locator polynomial $\lambda(x)$ computed by the iBM machine (`ld = 0`). The actual search is then started (`ld = 1`). At each clock cycle, the block performs h parallel evaluations of $\lambda(x)$ in GF(2^m) and outputs a $h-$bit word, denoted as `errmask`. Each bit of `errmask` corresponds to one of the h candidate error locations that have been evaluated. Asserted bits denote detected errors. This mask can then be XORed (outside the Chien Machine) with the related bits of the codeword in order to correct the detected erroneous bits.

The architecture of Fig. 4.11 provides an adaptable Chien machine with lower area consumption than other designs [25], having, at the same time, a marginal impact on performance. Four interesting features contribute to such optimization: (i) constant multipliers substructure sharing, (ii) adaptability to the correction capability, (iii) improved fast skipping to reduce the decoding time, and (iv) reduced full GF multipliers area. In the sequel, we briefly address each feature.

The first feature is represented by the optimized GF Constant Multipliers (optGFCM) networks of Fig. 4.11. The h parallel evaluations are based on equation (4.4). In the worst case ($t = t_M$), the parallel evaluation of equation (4.4) requires a matrix of $t_M \times h$

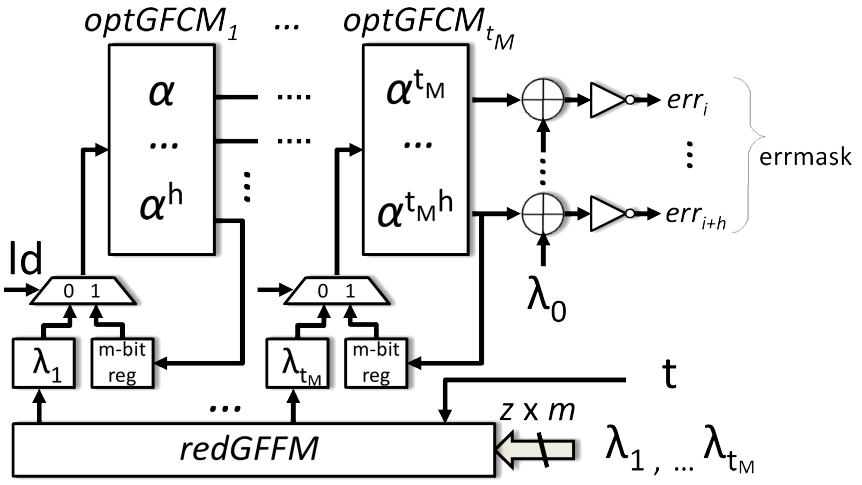


Figure 4.11: Architecture of the proposed parallel adaptable Chien Machine with parallelism equal to h

constant Galois multipliers. They multiply the content of the t_M registers by $\alpha, \alpha^2, ..., \alpha^{t_M}$, respectively. However, we can note that each column of constant GF multipliers shares the same multiplicand. Therefore, we can iteratively group their best-matching combinations [26] into the t_M optGFCM networks of Fig. 4.11. Such optGFCMs provide up to 60% reduction of the hardware complexity of the machine with no impact on performance.

The second feature is the adaptability of the Chien machine. The rows of the matrix define the parallelism of the block (i.e., the number of evaluations per clock cycles), while the columns define the maximum correction capability of the block. Whenever the selected correction capability t is lower than t_M, the coefficients of the error locator polynomial of degree greater than t are equal to zero and do not contribute to equation (4.4), thus allowing us to adapt the computation to the different correction capabilities.

The third feature stems from a simple observation. Depending on the selected correction capability t, not all the elements of GF(2^m) represent realistic error locations. In fact, considering a codeword composed of k bits of the original message and $r = m \cdot t$ parity bits, only $k + m \cdot t$ out of 2^m elements of the Galois field represent realistic error locations. Given that an error location L is the inverse of the related GF element ($L = 2^m - 1 - i$), the elements of GF(2^m) in which the error locator polynomial must be evaluated are in the

following range:

$$\left[\underbrace{\alpha^{2^m-1}}_{\text{error location L=0}} \quad , \quad \underbrace{\alpha^{2^m-k-m\cdot t}}_{\text{error location } L=k+m\cdot t-1} \right] \tag{4.9}$$

All elements between α^0 and $\alpha^{2^{m-k-m\cdot t}}$ can be skipped to reduce the computation time. Differently from fixed correction capability fast skipping Chien machines this interval is not constant here but depends on the selected t. The architecture of Fig. 4.11 implements an adaptable fast skipping by initializing the internal registers to the coefficients of the error corrector polynomial multiplied by a proper value $\beta^t_{ini} = \alpha^{2^m-k-m\cdot t-1}$. For each value of t, t_M $m-$bit constant values corresponding to β^t_{ini}, $\left(\beta^t_{ini}\right)^2$, ..., $\left(\beta^t_{ini}\right)^{t_M}$ must be stored in an internal ROM (not shown in Fig. 4.11) and multiplied by the coefficients λ_i using a full GF multiplier.

This is connected with the last feature, the reduced GF Full Multipliers (redGFFM) network of Fig. 4.11. Each full GF multiplier has a high cost in terms of area. Since they are used only during initialization of the Chien, the redGFFM adopts only $z \leqslant t_M$ full GF multipliers. It also includes a (λ) input port to input z coefficients, per clock cycles, of the error locator polynomial. This network enables to reduce area consumption, at a reasonable cost in terms of latency.

For the sake of brevity, a detailed description of the controller required to fully coordinate the decoder's modules interaction is out of the scope of this chapter.

4.6 Experimental Results

This section provides experimental data from the implementation of the adaptable BCH codec proposed on a selected case study.

4.6.1 Automatic generation framework

To cope with the complexity of a manual design of these blocks, a semi-automatic generation tool named ADAGE (ADaptive ECC Automatic GEnerator) [44] able to generate a fully synthesizable adaptable BCH codec core following the proposed architecture has been designed and exploited in this experimentation extending a preliminary framework previously introduced in [19]. The overall architecture of the framework is in Fig. 4.12.

The code analyzer block represents the first computational step required to select the desired code correction capability based on the Bit Error Rate (BER) of a page of the se-

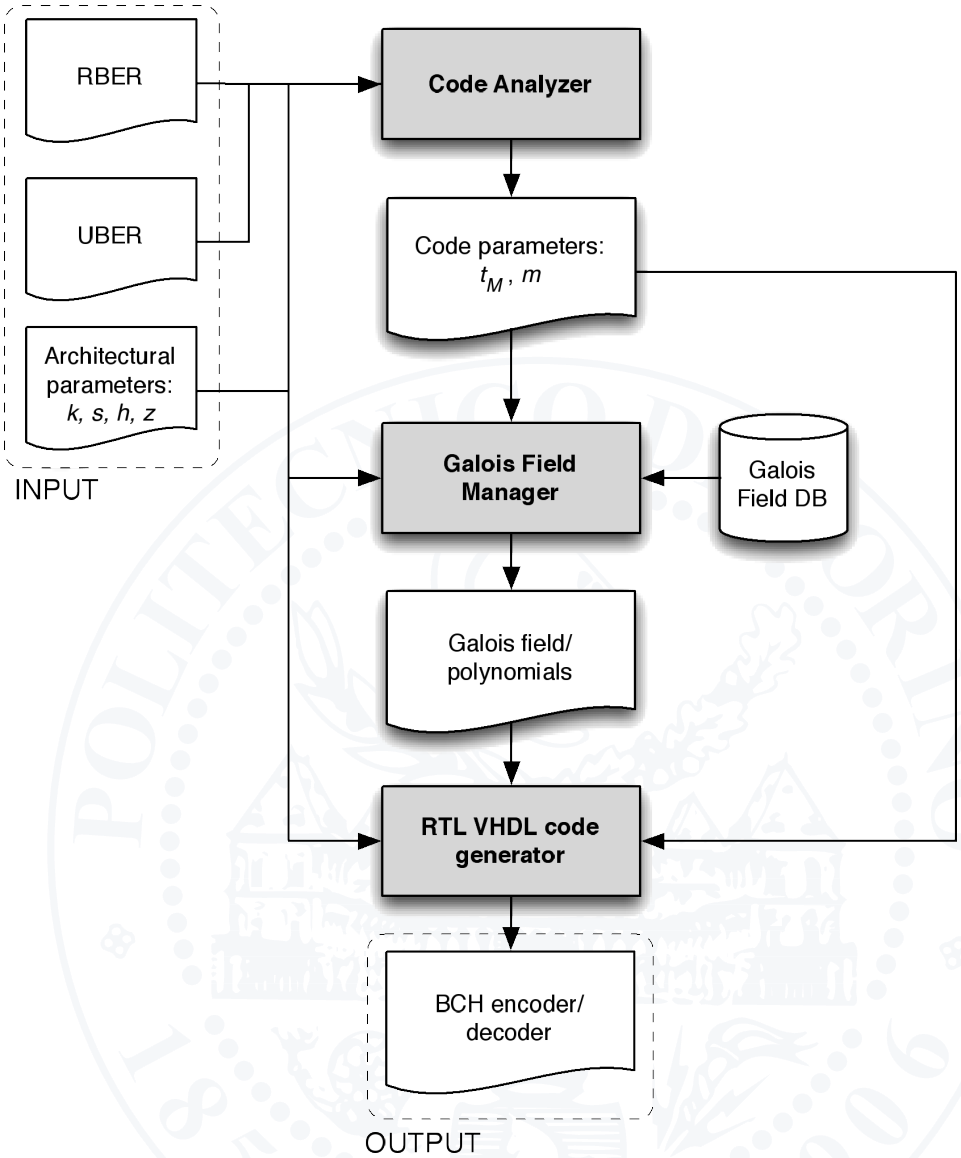


Figure 4.12: BCH codec automatic generation framework.

lected flash [90]. The BER is the fraction of erroneous bits of the flash. It is the key factor used to select the correction capability. Two values of BER must be considered. The former is the raw bit error rate (RBER), i.e., the BER before applying the error correction. It is technology/environment dependent and increases with the aging of the page [13, 141]. The latter is the uncorrectable bit error rate (UBER), i.e., the BER after the application of the ECC, which is application dependent. It is computed as the probability of having more than t errors in the codeword (calculated as a binomial distribution of randomly occurred bit errors) divided by the length of the codeword [34]:

$$UBER = \frac{P(E > t)}{n} = \frac{1}{n} \sum_{i=t+1}^{n} \binom{n}{i} \cdot RBER^{i} \cdot (1 - RBER)^{n-i} \tag{4.10}$$

Given the RBER of the flash and the target UBER, Eq. 4.10 can be exploited to compute the maximum required correction capability of the code and consequently the value of m that defines the target GF. Given these two parameters, the Galois Field manager exploits an internal polynomials database to generate the set of minimal polynomials and the related generator polynomials for the selected code.

Finally, the RTL VHDL code generator combines these parameters and generates a RTL description of the BCH encoder and decoder implementing the architecture illustrated in this paper.

The whole framework combines Matlab software modules with custom C programs. The full framework code is available for download at `http://www.testgroup.polito.it` in the Tools section of the website.

4.6.2 Experimental setup

Experiments have been performed, using as a case study a 2-bit per cell MLC NAND Flash Memory featuring a 45nm manufacturing process designed for low-power applications, with page size of 2KB plus 64B of spare cells. The memory has an 8-bit I/O interface. Considering the design of the BCH code, the current trend is to enlarge the block size k over which ECC operations are performed. In fact, longer blocks better handle higher concentrations of errors, providing more protection while using fewer parity bits [42]. For this reason, we adopted a block size k = 2KB, equal to the page size of the selected memory.

Experiments performed on the flash provided that, in a range between 10 and 100,000 program/erase (P/E) cycles on a page, the estimated RBER changes in a range $\left[9 \times 10^{-6} \div 3.5 \times 10^{-4}\right]$

[142]. With a target UBER of 10^{-13}, which is typical for commercial applications [55, 90], according to equation (4.10) we need to design a codec with correction capability in the range $t_{min} = 5$ up to $t_M = 24$. Since $k = 2^{14}$ and $t_M = 24$, from the expression $k + m \cdot t_M \leq 2^m - 1$ we deduce $m = 15$, thus obtaining a maximum of $r = m \cdot t_M \backsimeq 45$B of parity information. Given the 8-bit I/O interface of the memory, both the encoder and the decoder have been designed with an input parallelism of $s = 8$ bits. The values of h and z of the Chien Machine are a trade-off between the complexity of the decoder and the decoding time. Given the I/O parallelism of the flash and the area optimizations of Fig. 4.11, we opted for a Chien machine with parallelism $h = 8$ and $z = 1$ full GF multipliers.

In this experimentation we analyzed the three architectures summarized in Table 4.3.

Arch. 1 is classic BCH architecture with fixed correction capability of 24 errors per page. It represents the reference to compare our adaptable architectures.

Arch. 2 is an adaptable architecture with $t_{min} = 5 < t \leq 24$ using a traditional PPLFSR for the encoder and 24 PLFSRs for the syndrome calculation. It is worth mentioning here that, differently from what reported in the previous sections, the minimum required correction capability of the codec is higher than 1. This allows us to save space in the encoder PPLFSR since less polynomials must be stored, and in the Chien Machine's ROM since less β_{ini} terms must be stored.

Arch. 3 is an optimized version of Arch. 2 exploiting the use of a shOPPLFSR shared between the encoder and the decoder, to trade-off design complexity and decoding time. In order to optimize the use of the shOPPLFSR, we exploited the algorithm proposed in Section 4.3.1. Given our adaptable BCH code, a set of ad-hoc Matlab simulation scripts implement this preliminary analysis of 1,800[1] set Ω_i of polynomials. Each set Ω_i contains $t_M - t_{min} - 1 = 20$ generator polynomials required in the encoder and $t_M = 24$ minimal polynomials required in the decoder. This analysis aimed at finding the most suitable set of shared generator and minimal polynomials to trade-off between decoder's area and latency. A reasonable trade-off has been found using a shOPPLFSR composed of $\mathtt{N} = 5$ OPPLFSRs, each of which dividing by the following set of polynomials: $\{g_5, \psi_{29}, \psi_{39}\}$, $\{g_6, \psi_{31}, \psi_{41}\}$, $\{g_7, \psi_{33}, \psi_{43}\}$, $\{g_8, \psi_{35}, \psi_{45}\}$, and $\{g_9, ..., g_{24}, \psi_{37}, \psi_{47}\}$. The reader may refer to the appendix of this chapter for the full list of employed polynomials. All other structures remain almost unchanged. The comparison between Arch.1 and Arch. 2 enables to highlight the benefits of using an adaptable codec, while the com-

[1] our BCH code has 1,800 primitive polynomials $\psi_1(x)$

parison between Arch. 2 and Arch. 3 shows the advantages of adding optimized shared blocks.

Table 4.3: Characteristics of the analyzed architectures

	Adaptable	**OPPLFSRs**	**Chien Machine**
Arch. 1	No	-	$h = 8,\ t = 24$
Arch. 2	Yes	-	$h = 8,\ t \in [5,24]$
Arch. 3	Yes	5	$h = 8,\ t \in [5,24]$

4.6.3 Performance evaluations

Table 4.4 summarizes the main implementation details of the three selected architectures in terms of required parity bits and worst case encoding/decoding latency, expressed in terms of clock cycles.

Let us start with the evaluation of the amount of redundancy introduced by the two architectures. Arch. 1, which has a fixed correction capability of 24 errors per page, requires to store $m \cdot t_M = 24 \cdot 15 = 360$ parity bits (about 45B) for each 2KB page of the flash. This accounts for about 70% of the full spare area available for each page. Since the spare area cannot be fully reserved for storing ECC information (high-level functions, such as file system management and wear-leveling need to save considerable amount of information in this area), this percentage represents a considerable overhead for the selected device. Based on the results of Table 4.4, Fig. 4.13 shows how, for the adaptable codecs of both Arch. 2 and Arch. 3, the percentage of spare area dedicated for storing parity bits changes with the selected correction capability. The total occupation ranges in this case from 15% to 70% of the total spare area. This mitigates the overhead for storing parity bits whenever the error rate enables to select low correction capabilities (e.g., for devices in the early stages of their life).

For all implementations, the encoding latency depends on the size of the incoming message and is therefore constant regardless the adaptability of the encoder (see Table 4.4). The decoding latency is instead influenced by the correction capability, as reported in Table 4.4. Fig. 4.14 compares the decoding latency of the three architectures for each considered correction capability. Results are provided in number of clock cycles. It is worth mentioning here that timing estimations of Table 4.4 and Fig. 4.14 depict the worst-case scenario in which the Chien Machine must search all possible positions prior

Table 4.4: Worst case Parity Bits and Encoding/Decoding Latency. sh_{poly} denotes the maximum number of minimal polynomials shared in the shOPPLFSR of the syndrome machine

	Correction Capability	Parity Bits	Encoding latency (#Clk cycles)	Decoding latency (#Clk cycles)		
				Syndrome	iBM	Chien
		$m \cdot t$	$\frac{k}{s}$	$sh_{poly} \cdot \frac{k+mt}{s}$	$2t^2$	$\frac{h}{z} + \frac{k+mt}{h}$
Arch. 1	$t = 24$	360	2,048	2,093	1152	2,093
Arch. 2	$t = \{5,6,...,24\}$	$15 \cdot t$	2,048	$\frac{2{,}048 \cdot 8 + 15 \cdot t}{8}$	$2t^2$	$\frac{2{,}048 \cdot 8 + 15 \cdot t}{8}$
Arch. 3	$t = \{5,6,...,24\}$	$15 \cdot t$	2,048	$\frac{2 \times (2{,}048 \cdot 8 + 15 \cdot t)}{8}$	$2t^2$	$8 + \frac{2{,}048 \cdot 8 + 15 \cdot t}{8}$

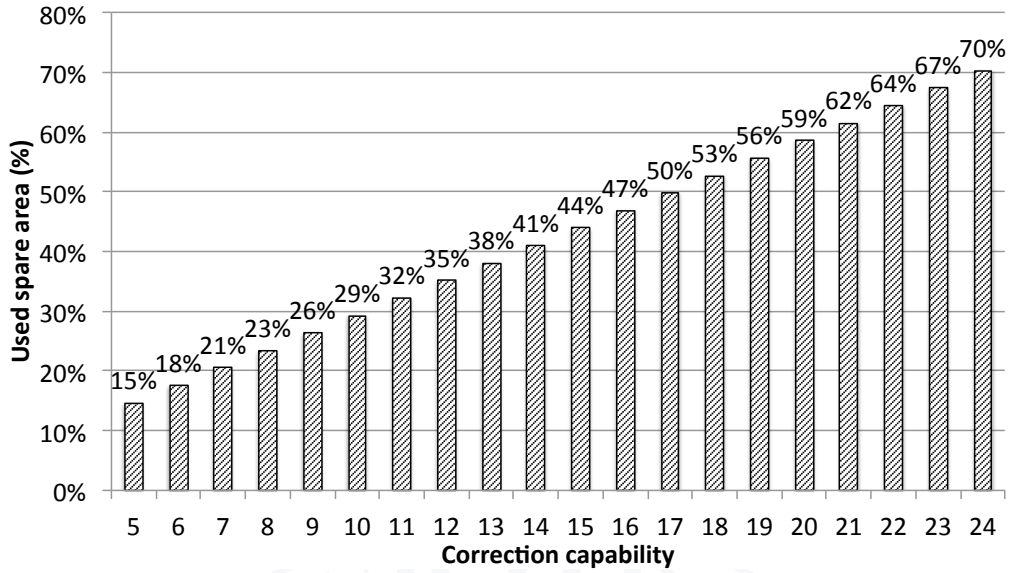


Figure 4.13: Percentage of spare area dedicated to parity bits while changing the correction capability of the adaptable codec of Arch. 2 and Arch. 3

to find the detected number of errors. Fig. 4.14 highlights that, for the lowest correction capability, both Arch. 2 and Arch. 3 enable 22% of decoding time reduction when compared to the fixed decoding time of Arch. 1. The decoding time increases with the correction capability. For Arch. 2, it reaches the same level of the fixed architecture when the correction capability reaches $t = 24$. Arch. 3 deviates from this behavior for $t \geqslant 20$. This penalty is introduced by the use of the shOPPLFSR in the Syndrome Machine. In this case, the codec includes 5 blocks to perform remainder computation with 10 minimal polynomials $\{\psi_{29}, \psi_{39}, \psi_{31}, \psi_{41}, \psi_{33}, \psi_{43}, \psi_{35}, \psi_{45}, \psi_{37}, \psi_{47}\}$. This implies doubling the syndrome computation time every time the required correction capability reaches a level in which all these polynomials must be used. Nevertheless, we will show that this reduced performance is counterbalanced by a reduced area overhead.

4.6.4 Synthesis Results

Synopsys Design Vision and a CORE 45nm technology cell library have been exploited to synthesize the designs. Table 4.5 shows the results of the synthesis of the three architectures. The hardware structures required to obtain the adaptability of the code introduce a certain area overhead. Considering Arch. 2, the area of the encoder increases since 19

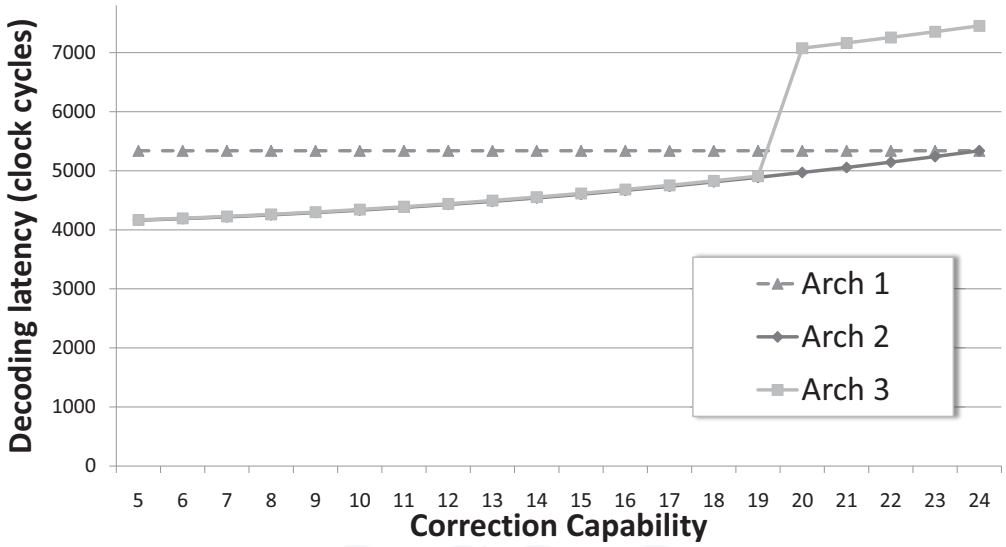


Figure 4.14: Worst case decoding latency for the three architectures considered.

generator polynomials must be stored in its ROM, while the area of the decoder increases due both to the aligners in the syndrome machine and to the ROM in the Chien machine to adapt the fast skipping process. Nevertheless, the introduced overhead is about 14% which is still acceptable. Considering Arch. 3, the introduced overhead is halved w.r.t. Arch. 2. The area of the encoder is almost comparable with Arch. 2. However, it now includes the shOPPLFSR and a smaller ROMs which contribute, with the LFSR sharing, at decreasing the area of the decoder. For both architectures we obtained a maximum clock frequency of 100MHz, which confirms that the adaptability does not impact the maximum speed of the circuit. This area result is interesting if compared with an estimation of the area for the adaptable architecture proposed in [25]. [25] designed a codec working on blocks of data of 512B, smaller than the 2KB used in this paper. Given the same maximum correction capability ($t_M = 24$), [25] uses a code defined on $GF(2^{13})$ instead of the code defined on $GF(2^{15})$ used in this paper. However, even if the code is simpler and the number of correction modes is smaller (only 4 correction modes), the area of the codec accounts about 158.9K equivalent gates[2], which is higher than the 111.4K and the 105.2K equivalent gates of the Arch. 2 and Arch. 3 proposed.

Fig. 4.15 compares the decoder's dynamic power dissipation of the three architectures computed using Synopsys PrimeTime. As for the decoding latency the analysis has

[2] Equivalent gates for state-of-the-art architectures have been estimated from the information provided in the papers

Table 4.5: Synthesis Results

	Comp.	Max Clock	Equiv. Gates	Over-head
Arch. 1	Encoder	100 MHz	33.3 K	
	Decoder	100 MHz	64.1 K	
	Overall	100 MHz	97.4 K	(ref.)
Arch. 2	Encoder	100 MHz	40.8 K	
	Decoder	100 MHz	70.6 K	
	Overall	100 MHz	111.4 K	14%
Arch. 3	Encoder	100 MHz	39.2 K	
	Decoder	100 MHz	66.0 K	
	Overall	100 MHz	105.2 K	7%

been performed for a worst-case simulation in which t errors are injected at the end of the codeword so that the Chien Machine must search all possible positions prior to detect all errors. Considering Arch. 2, results show that the introduction of the adaptability enables up to 15% of dynamic power saving when the lowest correction capability can be selected. This is due to the fact that the portions of the circuits not required for low correction capabilities are disabled. The introduction of the optimizations proposed in Arch. 3 has no significant impact on the dynamic power that remains almost equal to the one of Arch. 2.

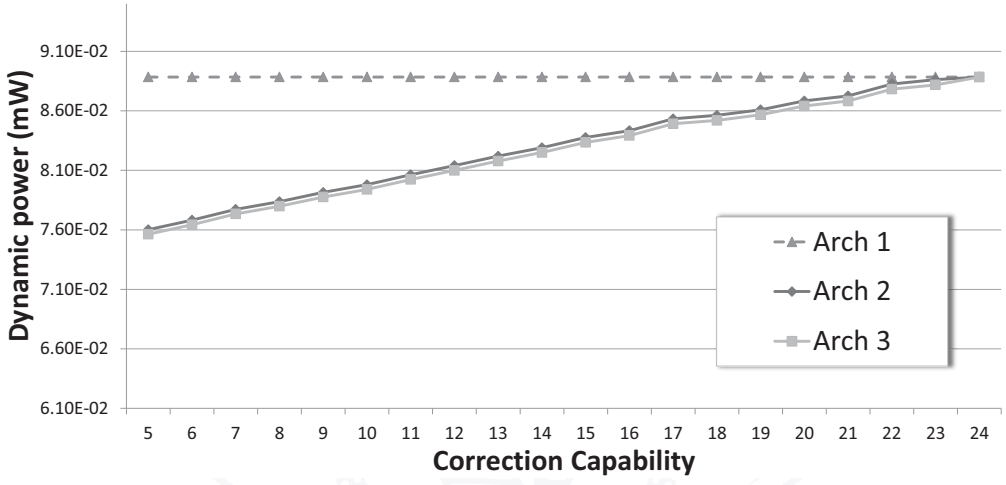


Figure 4.15: Worst case dynamic power consumption of the three decoders for the three considered architectures. Power is expressed in mW.

4.7 A Cross-Layer Approach for New Reliability-Performance Trade-Offs in MLC NAND Flash Memories

In spite of the mature cell structure, the memory controller architecture of MLC NAND Flash memories is evolving fast in an attempt to improve the uncorrected/miscorrected bit error rate (UBER) and to provide a more flexible usage model where the performance-reliability trade-off point can be adjusted at runtime. However, optimization techniques in the memory controller architecture cannot avoid a strict trade-off between Uncorrected BER (UBER) and read throughput. In collaboration with the Universita' di Ferrara, we show that co-optimizing ECC architecture configuration in the memory controller with program algorithm selection at the technology layer, a more flexible memory sub-system arises, which is capable of unprecedented trade-offs points between performance and reliability. For sake of brevity, this is out of the scope of this chapter. The interested reader may refer to [142] for more details about this topic.

4.8 Conclusions

This chapter proposed a BCH codec architectures and its related automatic generation framework which enables its code correction capability to be selected in a predefined range of values. Designing an ECC system whose correction capability can be modified in-field has the potentiality to adapt the correction schema to the reliability requirements the flash encounters during its life-time, thus maximizing performance and reliability.

Experimental results on a selected NAND flash memory architecture proved that the proposed solution reduces spare area usage, decoding time, and power dissipation whenever small correction capability can be selected.

Ongoing research is currently focusing on the definition of an efficient heuristic to compute, at run-time, the best correction capability that must be applied to a page of the flash to adapt the code to the instantaneous error rate of the device. Such an heuristic could be coupled with the hardware architecture proposed in this chapter and integrated at the file system level to obtain a full adaptive flash based storage system.

Table 4.6: Minimal polynomials expressed with the corresponding hexadecimal string of coefficients

ψ_1	0x F465	ψ_{17}	0x B13D	ψ_{33}	0x 8011
ψ_3	0x C209	ψ_{19}	0x B305	ψ_{35}	0x BA2B
ψ_5	0x B3B7	ψ_{21}	0x A495	ψ_{37}	0x D95F
ψ_7	0x E6EB	ψ_{23}	0x 88C7	ψ_{39}	0x BFF5
ψ_9	0x E647	ψ_{25}	0x C357	ψ_{41}	0x BA87
ψ_{11}	0x D4E5	ψ_{27}	0x B2C1	ψ_{43}	0x 9BEB
ψ_{13}	0x 8371	ψ_{29}	0x 97DD	ψ_{45}	0x 93CB
ψ_{15}	0x EDD9	ψ_{31}	0x FA49	ψ_{47}	0x F385

SUMMARY

The strong transistor miniaturization and the adoption of an increasing number of levels per cell require ECCs to be systematically applied to NAND flash. Choosing the correction capability of an ECC is a trade-off between reliability and code complexity. We therefore designed a BCH system whose correction capability can be modified in-field. In fact, it is an attractive solution to adapt the correction schema to the reliability requirements the flash encounters during its life-time, thus maximizing performance and reliability. Experimental results on a selected NAND flash memory architecture have been able to show that the proposed codec enables to reduce spare area usage, decoding time and power dissipation.

The whole design process was supported by the novel ADaptive ECC Automatic GEnerator (ADAGE) design environment. ADAGE is a fully customizable tool aimed at automatic generation of adaptable BCH architectures. ADAGE is able to automatically generate the VHDL code of the designed adaptable BCH-based architecture. Such a code can be thoroughly simulated, validated and synthesized on ASIC or FPGA. Experimental results confirmed the efficiency of our ADAGE tool.

Ongoing research is currently focusing on the definition of an efficient heuristic to compute, at run-time, the best correction capability that must be applied to a page of the flash to adapt the code to the instantaneous error rate of the device. Such an heuristic could be coupled with the hardware architecture proposed in this chapter and integrated into an open-source flash memory file system in order to test its efficiency in a real working environment.

Table 4.7: Generator polynomial expressed with the corresponding hexadecimal string of coefficients

g_5	0x 0163C68D766635253
g_6	0x 018FBE36E3B716D8BCE32
g_7	0x 01E573FBB06E46A828C1C770C
g_8	0x 01F28E94D9B550543AC42286CF418
g_9	0x 01D6634FC565E6012E441926C07B8D59
g_{10}	0x 018B24C1E935C04DC6BC73E0BDB98405C4EA
g_{11}	0x 01E8B4BA11F717E75A1F5E0EC4FBCD65DA8FFF24
g_{12}	0x 018FB50FA2969CDC5EAFA1C24BD9E5AA92A2227EC668
g_{13}	0x 012E919C715C15310DA7103C0AB656C7FE330613197631D
g_{14}	0x 01E59154D4757E35CBDCE8247F4686EACC2C96C8209D848BDCE
g_{15}	0x 01E12C4539A437988318B8B0A756426E93CD50010 31DCB5DC430A0C
g_{16}	0x 01BE62D0F7C4D16FCCDD3CE20D7998280B591702D452F3541A51DA955D8
g_{17}	0x 019755B57BEBA0DD4C284FE4B4F4549C194CA6E75E542322123EAB270447821712
g_{18}	0x 016240D5F338473A9653892D4DDC334AEF9FE78E9B835C10D1C9106B14AA4AB4BD5CD4
g_{19}	0x 01B54AFF801C5EBB55EA214ADCCB051347A16418268264264299431B25E5B7CE34F402D938
g_{20}	0x 01CA788668B1303E48C4A41BE62900685C4A42DB04E267A642AC828841761945 01F076D19CF53
g_{21}	0x 015E830624B4D708788177787CA2DC6C89F7558E799E84DD10 27034F4DEC7476ADA565B11240FB4EE
g_{22}	0x 01D6ECB0041A40258ADA46542DB3657CFA042227D7CAADD770809ACC680C2886C0EACDC8D81D34565F7FC
g_{23}	0x 0102924C5CEA2B43968EFF54D1E0FAB54DEBFDC54428EDAE6FE2EE724B79CBC072C19CEB766864091E5551A38
g_{24}	0x 0141AE1262150974 03F13F41BE936020FAA0D6D486AD40BE0BED62DC87C4D8CF945A4D2A804411217E82829127AD

POLITECNICO DI TORINO
1859
1906

Those parts of the system that you can hit with a hammer (not advised) are called hardware; those program instructions that you can only *curse at* are called software.

Unknown

CHAPTER 5

SOFTWARE MANAGEMENT OF NAND FLASH MEMORY: ISSUES AND CHALLENGES

Contents of this chapter

5.1 File systems for flash memories [47]
5.2 Comparisons of the presented FFS [47]
5.3 FLARE: a Design Environment for Flash-based Space Applications [17, 50]
5.4 Wear Leveling Strategies: An Example

Among the different issues to consider when designing a flash-based mass storage system, the file management represents a challenging problem to address [47]. In fact, flash memories store and access data in a completely different manner if compared to magnetic disks. This must be considered at the OS level to grant existing applications an efficient access to the stored information. Two main approaches are pursuit by OSs and flash memory designers: (i) block-device emulation, and (ii) development of native file systems optimized to operate with flash-based devices [22].

Block-device emulation refers to the development of a hardware/software layer (i.e., Flash Translation Layer (FTL)) able to emulate the behavior of a traditional block device such as a hard-disk, allowing the OS to communicate with the flash using the same primitives exploited to communicate with magnetic-disks. The main advantage of this approach is the possibility of reusing available file systems (e.g., FAT, NTFS, ext2) to ac-

cess the information stored in the flash, allowing maximum compatibility with minimum intervention on the OS. However, traditional file systems do not take into account the specific peculiarities of the flash memories, and the emulation layer alone may be not enough to guarantee maximum performance.

The alternative to the block-device emulation is to exploit the hardware features of the flash device in the development of a *native Flash File System (*FFS*)*. An end-to-end flash-friendly solution can be more efficient than stacking a file system designed for the characteristics of magnetic hard-disks on top of a device driver designed to emulate disks using flash memories [52]. For efficiency reasons, this approach is becoming the preferred solution whenever embedded NAND flash memories are massively exploited.

The literature is rich of strategies involving block-device emulation, while, to the best of our knowledge, a comprehensive comparison of available native file systems is still missing. This chapter discusses how to properly address the issues of using NAND flash memories as mass-memory devices from the native file system standpoint. We hope that the ideas and the solutions proposed in this chapter will be a valuable starting point for designers of NAND flash-based mass-memory devices.

5.1 File systems for flash memories

As shortly described in the introduction of this chapter, at the OS level the common alternatives to manage flash based mass-storage devices are block-device emulation and native flash file systems [22]. Both approaches try to address the issues discussed in Section 2.1. Fig. 5.1 shows how the two solutions can be mapped in a generic OS.

The block-device emulation approach hides the presence of a flash memory device, by emulating the behavior of a traditional magnetic hard-disk. The flash device is seen as a contiguous array of storage blocks. This emulation mechanism is achieved by inserting at the OS level a new layer, referred to as FTL. Different implementations of FTL have been proposed [23, 61, 66]. The advantage of using an FTL is that existing file systems, such as NTFS, ext2 and FAT, usually supported by the majority of modern OSs, can be directly used to store information in the flash. However, this approach has many performance restrictions. In fact, existing file systems do not take into account the critical issues imposed by the flash technology (see Section 2.1) and in several situations they may behave in contrast with these constraints. Very sophisticated FTL must be therefore designed with heavy consequences on the performance of the system. Moreover, the

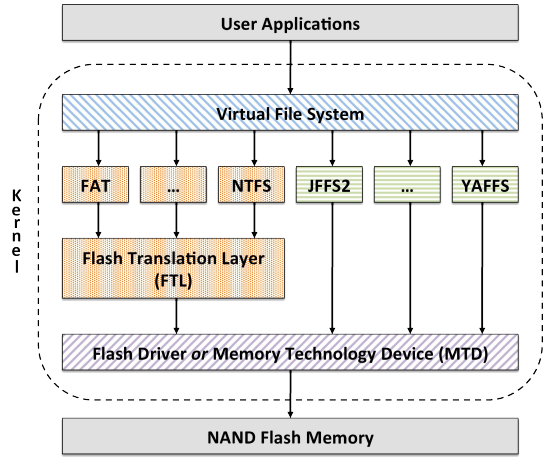


Figure 5.1: Flash Translation Layer and Flash File Systems

typical block size managed by traditional file systems usually does not match the block size of a flash memory. This imposes the implementation of complex mechanisms to properly manage write operations [52].

The alternative solution, to overcome the limitation of using an FTL, is to expose the hardware characteristics of the flash memory to the file system layer, demanding to this module the full management of the device. These new file systems, specifically designed to work with flash memories, are usually referred to as Flash File System (FFS). This approach allows the file system to fully exploit the potentiality of a flash memory guaranteeing increased performance, reliability and endurance of the device. In other words, if efficiency is more important than compatibility, FFS is the best option to choose.

The way FFSs manage the information is somehow derived from the model of *journaled* file systems. In a journaled file system, each metadata modification is written into a journal (i.e., a log) before the actual block of data is modified. This in general helps recovering information in case of crash. In particular log-structured file systems [7, 110, 132] take the journaling approach to the limit since the journal *is* the file system. The disk is organized as a log consisting of fixed-sized segments of contiguous areas of the disk, chained together to form a linked list. Data and metadata are always written to the end of the log, never overwriting old data. Although this organization has been in general avoided for traditional magnetic disks, it perfectly fits the way information can be saved into a flash memory since data cannot be overwritten in these devices, and write operations must be performed on new pages. Furthermore, log-structuring the file

system on a flash does not influence the read performance as in traditional disks, since the access time on a flash is constant and does not depend on the position where the information is stored [52].

FFSs are nowadays mainly used whenever so called Memory Technology Device (MTD) is available in the system, i.e., embedded flash memories that do not have a dedicated hardware controller. Removable flash memory cards and USB flash drives are in general provided with a built-in controller that in fact behaves as an FTL and allows high compatibility and portability of the device. FFSs have therefore limited benefits on these devices.

Several FFSs are available. A possible approach to perform a taxonomy of the available FFSs is to split them into three categories: (i) experimental FFSs documented in scientific and technical publications, (ii) open source projects and (iii) proprietary products.

5.1.1 Flash file systems in the technical and scientific literature

Several publications proposed interesting solutions for implementing new FFSs [69, 72, 124, 135]. In general each of these solutions aims at optimizing a subset of the issues proposed in Section 2.1.

Although these publications in general concentrate on algorithmic aspects, and provide reduced information about the real implementation, they represent a good starting point to understand how specific problems can be solved in the implementation of a new FFS.

5.1.1.1 eNVy

Fig. 5.2 describes the architecture of a system based on eNVy, a large non-volatile main memory storage system built to work with flash memories [135].

The main goal of eNVy is to present the flash memory to a host computer as a simple linear array of non-volatile memory. The additional goal is to guarantee an access time to the memory array as close as possible to those of an SRAM (about 100us) [52].

The reader may refer to [134] for a complete description of the eNVy FFS.

Technology eNVy adopts an SLC NAND flash memory with page size of 256B.

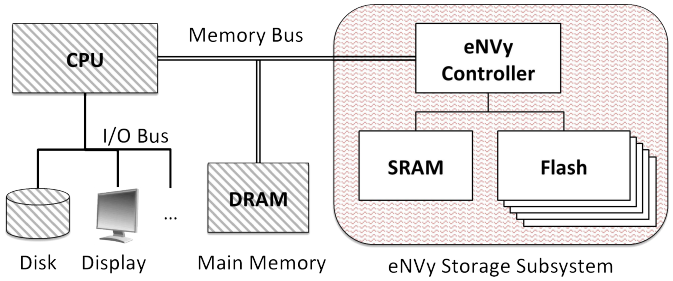


Figure 5.2: Architecture of eNVy

Architecture The eNVy architecture combines an SLC NAND flash memory with a small and fast battery-backed Static RAM (SRAM). This small SRAM is used as a very fast write buffer required to implement an efficient copy-on-write strategy.

Address translation The physical address space is partitioned into pages of 256B that are mapped to the pages of the flash. A page table stored in the SRAM maintains the mapping between the linear logical address space presented to the host and the physical address space of the flash. When performing a write operation, the target flash page is copied into the SRAM (if not already loaded), the page table is updated and the actual write request is performed into this fast memory. As long as the page is mapped into the SRAM, further read and write requests are performed directly using this buffer. The SRAM is managed as a First In First Out (FIFO), new pages are inserted at the end, while pages are flushed from the tail when their number exceeds a certain threshold [52].

Garbage collection When the SRAM write buffer is full, eNVy attempts to flush pages from the SRAM to the flash. This in turn requires to allocate a set of free pages in the flash. If there is no free space, the eNVy controller starts a garbage collection process called cleaning in the eNVy terminology (see Fig. 5.3).

When eNVy cleans a block (segment in the eNVy terminology), all of its live data (i.e., valid pages) are copied into an empty block. The original block is then erased and reused. The new block will contain a cluster of valid pages at its head, while the remaining space will be ready to accept new pages. A clean (i.e., completely erased) block must be always available for the next cleaning operation.

The policy for deciding which block to clean is an hybrid between a *greedy* and a *locality gathering* method. Both methods are based on the concept of "flash cleaning cost",

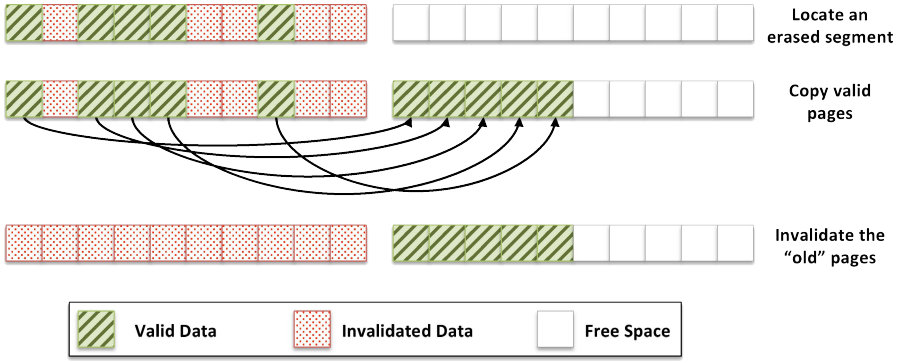


Figure 5.3: Steps of the eNVy cleaning process

defined as $\frac{\mu}{1-\mu}$ where μ is the utilization of the block. Since after about 80% utilization the cleaning cost reaches unreasonable levels, μ in cannot exceed this threshold.

The *greedy* method cleans the block with the majority of invalidated pages in order to maximize the recovered space. This method lowers cleaning costs for uniform distributions (i.e., it tends to clean blocks in a FIFO order), but performance suffers as the locality of references increases.

The *locality gathering* algorithm attempts to take advantage from high locality of references. Since hot blocks are cleaned more often than cold blocks, their cleaning cost can be lowered by redistributing data among blocks. However, for uniform access distributions, this technique prevents cleaning performance from being improved. In fact, if all data are accessed with the same frequency, the data distribution procedure allocates the same amount of data to each segment. Since pages are flushed back to their original segments to preserve locality, all blocks always stay at μ = 80% utilization, leading to a fixed cleaning cost of 4.

eNVy adopts an hybrid approach, which combines the good performance of the FIFO algorithm for uniform access distributions and the good results of the locality gathering algorithm for higher locality of references.

The high performance of the system is guaranteed by adopting a wide bus between the flash and the internal RAM, and by temporarily buffering accessed flash pages. The wide bus allows pages stored in the flash to be transferred to the RAM in one cycle, while buffering pages in RAM allows to perform several updates to a single page with a single RAM-to-flash page transfer. Reducing the number of flash writes reduces the number of unit erasures, thereby improving performance and extending the lifetime of the device

[52].

However, using a wide bus has a significant drawback. To build a wide bus, several flash chips are used in parallel [135]. This increases the effective size of each erase unit. Large erase units are harder to manage and, as a result, they are prone to accelerated wear [52]. Finally, although [135] states that a cleaning algorithm is designed to evenly wear the memory and to extend its lifetime, the work does not present any explicit wear leveling algorithm. The bad block management and the ECC strategies are missing as well.

5.1.1.2 Core flash file system (CFFS)

[124] proposes the Core Flash File System (CFFS) for NAND flash-based devices. CFFS is specifically designed to improve the booting time and to reduce the garbage collection overhead.

The reader may refer to [124] for a complete description of CFFS. While concentrating on boot time and garbage collection optimizations, the work neither presents any explicit bad block management nor any error correction code strategy.

Address translation CFFS is a log-structured file system. Information items about each file (e.g., file name, file size, timestamps, file modes, index of pages where data are allocated, etc.) are saved into a special data structure called inode. Two solutions can be in general adopted to store inodes in the flash: (i) storing several inodes per page, thus optimizing the available space, or (ii) storing a single inode per page. CFFS adopts the second solution. Storing a single inode per page introduces a certain overhead in terms of flash occupation, but, at the same time, it guarantees enough space to store the index of pages composing a file, thus reducing the flash scan time at the boot.

CFFS classifies inodes in two classes as reported in Fig. 5.4. *i-class1* maintains direct indexing for all index entries except the final one, while *i-class2* maintains indirect indexing for all index entries except the final one. The final index entry is indirectly indexed for i-class1 and double indirectly indexed for i-class2. This classification impacts the file size range allowed by the file system. Let us assume to have 256B of metadata for each inode and a flash page size of 512B. The inode will therefore contain 256B available to store index pointers. A four-byte pointer is sufficient to point to an individual flash page. As a consequence, $256/4 = 64$ pointers fit the page. This leads to:

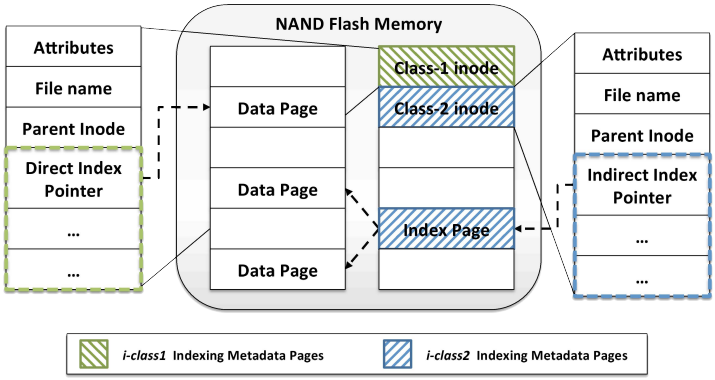


Figure 5.4: An example of direct (*i-class1*) and indirect (*i-class2*) indexing for a NAND flash

- ***i-class1***: 63 pages are directly indexed and 1 page is indirectly indexed, which in turn can directly index $512/4 = 128$ pages; as a consequence the maximum allowed file size is $(63+128) \times 512B = 96KB$

- ***i-class2***: 63 pages are indirectly indexed, each of which can directly index $512/4 = 128$ pages, thus they can address an overall amount of $63 \times 128 = 8064$ pages. 1 page is double indirectly indexed, which in turn can indirectly index up to $(512/4)^2 = 16384$ pages. Therefore, the maximum allowed file size is $(8064+16384) \times 512B = 12MB$

If the flash page is $2KB$, the maximum file size is $1916KB$ for i-class1 and $960MB$ for i-class2.

The reason CFFS classifies inodes into two types is the relationship between the file size and the file usage patterns. In fact, most files are small and most write accesses are to small files. However, most storage is also consumed by large files that are usually only accessed for reading [124]. The *i-class1* requires one additional page consumption for the inode[1], but can address only pretty small files. Each writing into an indirect indexing entry of *i-class2* causes the consumption of two additional pages, but it is able to address bigger files.

When a file is created in CFFS, the file is first set to *i-class1* and it is maintained in this state until all index entries are allocated. As the file size grows, the inode class is altered from *i-class1* to *i-class2*. As a consequence, most files are included in *i-class1*

[1] in general, the number of additional flash pages consumed due to updating the inode index information is proportional to the degree of the indexing level